SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 0(0), 1–18 (MONTH 1900)

Value Numbering

PRESTON BRIGGS
Tera Computer Company, 2815 Eastlake Avenue East, Seattle, WA 98102

AND

KEITH D. COOPER L. TAYLOR SIMPSON
Rice University, 6100 Main Street, Mail Stop 41, Houston, TX 77005

SUMMARY

Value numbering is a compiler-based program analysis method that allows redundant computations to be removed. This paper compares hash-based approaches derived from the classic local algorithm[1] with partitioning approaches based on the work of Alpern, Wegman, and Zadeck[2]. Historically, the hash-based algorithm has been applied to single basic blocks or extended basic blocks. We have improved the technique to operate over the routine's dominator tree. The partitioning approach partitions the values in the routine into congruence classes and removes computations when one congruent value dominates another. We have extended this technique to remove computations that define a value in the set of available expressions (AVAIL)[3]. Also, we are able to apply a version of Morel and Renvoise's partial redundancy elimination[4] to remove even more redundancies.

The paper presents a series of hash-based algorithms and a series of refinements to the partitioning technique. Within each series, it can be proved that each method discovers at least as many redundancies as its predecessors. Unfortunately, no such relationship exists between the hash-based and global techniques. On some programs, the hash-based techniques eliminate more redundancies than the partitioning techniques, while on others, partitioning wins. We experimentally compare the improvements made by these techniques when applied to real programs. These results will be useful for commercial compiler writers who wish to assess the potential impact of each technique before implementation.

KEY WORDS Code Optimization Value Numbering Redundancy Elimination

INTRODUCTION

Value numbering is a compiler-based program analysis 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. The primary objective of value numbering is to assign an identifying number (a *value number*) to each expression in a particular way. The number must have the property that two expressions have the same number if the compiler can prove they are equal for all possible program inputs. The numbers can then be used to find redundant computations and remove them. There are two other objectives accomplished by certain forms of value numbering:

1. To recognize certain algebraic identities, like $i = i + 0$ and $j = j \times 1$, and to use them to simplify the code and to expand the set of expressions known to be equal.

CCC 0038–0644/00/000001–18 *Received Sept 29, 1995*
 Revised October 21, 1997

2. To evaluate expressions whose operands are constants and to propagate their values through the code.

This paper describes different techniques for assigning numbers and handling redundancies. There are several ways to accomplish each of these goals, and the methods can be applied across different scopes. It includes an experimental evaluation of the relative effectiveness of these different approaches.

In value numbering, the compiler can only assign two expressions the same value 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 value number for the resulting expression. Hashing provides an efficient representation of the expressions known at any point during the analysis. The hash-based techniques are on-line methods that transform the program immediately. Their efficiency relies on the constant expected-time behavior of hashing.* This approach can easily be extended to propagate constants and simplify algebraic identities.
- 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 transforming 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[6]. The partitioning algorithm cannot propagate constants or simplify algebraic identities.

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 algorithm based on using a unified hash table for the entire procedure, (3) an extension of Alpern, Wegman, and Zadeck's partition-based global value numbering algorithm to perform AVAIL-based removal of expressions or partial redundancy elimination, and (4) an experimental comparison of these techniques in the context of an optimizing compiler.

HASH-BASED VALUE NUMBERING

Cocke and Schwartz[1] describe a local technique that uses hashing to discover redundant computations and fold constants. Each unique expression is identified by its *value number*. Two computations in a basic block have the same value number if they are provably equal. In the literature, this technique and its derivatives are called "value numbering."

Figure 1 shows high-level pseudo-code for value numbering single basic blocks. The algorithm uses two arrays to maintain a mapping between variable names and value numbers. The *VN* array maps variable names to value numbers, and the *name* array maps value numbers to variable names. For each instruction in the block, from top to bottom, we find the value numbers of the operands and hash the operator and the value numbers of the operands to obtain a unique number. If the value has already been computed in the block, it will already

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

```
for each assignment a of the form "x ← y op z" in block B
    expr ← ⟨VN[y] op VN[z]⟩
    if expr can be simplified to expr′
        Replace right-hand side of a with the simplified expression
        expr ← expr′
    if expr is found in the hash table with value number v
        VN[x] ← v
        if VN[name[v]] = v
            Replace right-hand side of a with name[v]
    else
        v ← next available value number
        VN[x] ← v
        Add expr to the hash table with value number v
        name[v] ← x
```

Figure 1. Basic-block value numbering

exist in the hash table. If the original variable still contains the same value, the recomputation can be replaced with a reference to that variable. To verify this condition, we look up the name corresponding to the value number, v, and verify that its value number is still v (*i.e.,* $VN[name[v]] = v$). 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 2. 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. We can handle this by making each entry in the *name* array contain a list of variable names and carefully keeping the lists up to date. At statement (2), the situation is worse; $X + Y$ is in the hash table, but it is not available anywhere.

	$A \leftarrow X + Y$	$A_0 \leftarrow X + Y$
	$B \leftarrow X + Y$	$B_0 \leftarrow X + Y$
	$A \leftarrow 1$	$A_1 \leftarrow 1$
(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)	$D \leftarrow X + Y$	$D_0 \leftarrow X + Y$
	Original	**SSA Form**

Figure 2. Value numbering example

As described, the technique operates over single basic blocks. With some minor modifications, we can apply it to an expanded scope, called an *extended basic block*[7]. An extended basic block is a sequence 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. To apply value numbering to a single extended basic block, we can simply apply the single block algorithm to each block in the sequence, in order, and use the results from B_{i-1} to initialize the tables for B_i. This works because each B_i has a single predecessor, $1 < i \leq n$.

If a block has multiple successors, then it may be a member of more than one extended basic block. For example, consider the if-then-else construct shown in Figure 3. Block B_1 is contained in two extended basic blocks: $\{B_1, B_2\}$ and $\{B_1, B_3\}$. These blocks are related by a common prefix. In fact, if the intersection of two extended basic blocks is non-empty, it must be a common prefix of both. Thus, a collection of extended basic blocks related by intersection forms a tree, and the trees form a forest representing the control-flow graph. The start block and each block with multiple predecessors correspond to the root of a tree, and each block with a single predecessor, p, is a child of p. We can use this tree structure to avoid processing any basic block more than once.

The tree representation of extended basic blocks leads to a straight forward and efficient technique for value numbering. It suggests that each tree should be processed in a preorder walk using a scoped hash table similar to one that would be used for processing declarations in a language with nested lexical scopes[7, 8]. At any point during the processing, the scoped table contains a sequence of nested scopes, one for each block that is an ancestor of the current block.

- As new blocks are processed, new scopes are created in the table. Any entries added to the current scope will supersede entries with the same name in any enclosing scope. Searches are performed starting with the innermost scope and proceeding outward until a matching entry is found.
- As the algorithm returns upward from a block, it must undo the effects of processing that block. Using a scoped table, this corresponds to deleting the block's scope from the table. It must also restore the entries in the *name* and *VN* arrays. In practice, this adds a fair amount of overhead and complication to the algorithm, but it does not change the asymptotic complexity.

The scoped table matches the tree structure of sets of related blocks. It lets the algorithm avoid reprocessing blocks that appear in multiple extended basic blocks. The next section shows how to use the properties of static single assignment form to eliminate the *name* array and to avoid the complication of restoring the *VN* array.

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[6]. The basic idea used in constructing SSA form is to give unique names to the targets of all assignments in the routine, and then to overwrite uses of the assignments with the new names. Special assignments, called ϕ-functions, are inserted to select the appropriate definition where more that one definition site (each with a unique SSA name) reaches a point in the routine. One ϕ-function is inserted at each join point for each name in the original routine. In practice, to save space and time, ϕ-functions are placed at only certain join points and for only certain names. Specifically, a ϕ-function is placed at the *birthpoint*[9] of each value – the earliest location

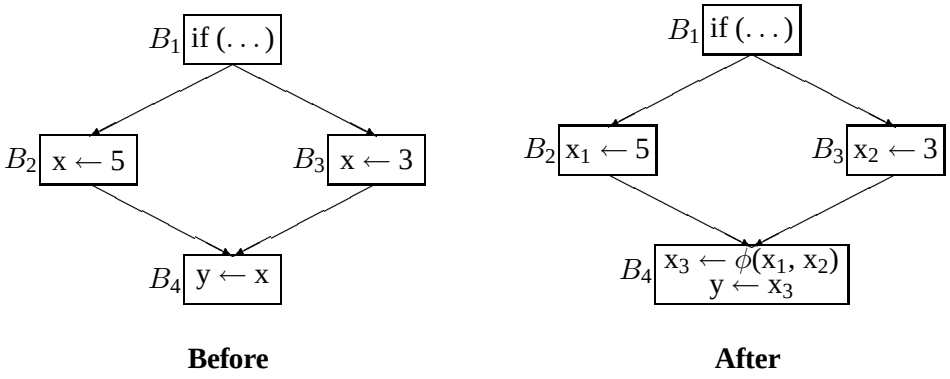

Figure 3. Conversion to SSA form

where the joined value exists. Each ϕ-function defines a new name for the original item as a function of all of the SSA names which are current at the end of the join point's predecessors. Any uses of the original name after the ϕ-function are replaced by the ϕ-function's result. The ϕ-function selects the value of the input that corresponds to the block from which control is transferred and assigns this value to the result.

The critical property of SSA for this work is the naming discipline that it imposes on the code. Each SSA name is assigned a value by exactly one operation in a routine; therefore, no name is ever reassigned, and no expression ever becomes inaccessible. The advantage of this approach becomes apparent if the code in Figure 2 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. We can also eliminate the *name* array, and we no longer need to restore the *VN* array.

Dominator-based value numbering technique

To extend the scope of optimization any further requires a mechanism for handling points in the control-flow graph where paths converge. The method for extended basic blocks already covers the maximal length regions without such merges. To handle merge points, we will rely on a well-understood idea from classic optimization and analysis—*dominance*. In a flow graph, if node X appears on every path from the start node to node Y, then X *dominates* Y ($X \geqslant Y$)[10]. If $X \geqslant Y$ and $X \neq Y$, then X *strictly dominates* Y ($X \gg Y$). The *immediate dominator* of Y (idom(Y)) is the closest strict dominator of Y[11]. In the routine's *dominator tree,* the parent of each node is its immediate dominator. Notice that all nodes that dominate a node X are ancestors of X in the dominator tree.

Aside from the naming discipline imposed, another key feature of SSA form is the information it provides about the way values flow into each basic block. A value can enter a block B in one of two ways: either it is defined by a ϕ-function at the start of B or it flows through B's parent in the dominator tree (*i.e.,* B's immediate dominator). These observations lead us

```
procedure DVNT(Block B)
    Mark the beginning of a new scope
    for each φ-function p of the form "n ← φ(...)" in B
        if p is meaningless or redundant
            Put the value number for p into VN[n]
            Remove p
        else
            VN[n] ← n
            Add p to the hash table
    for each assignment a of the form "x ← y op z" in B
        Overwrite y with VN[y] and z with VN[z]
        expr ← ⟨y op z⟩
        if expr can be simplified to expr'
            Replace a with "x ← expr'"
            expr ← expr'
        if expr is found in the hash table with value number v
            VN[x] ← v
            Remove a
        else
            VN[x] ← x
            Add expr to the hash table with value number x
    for each successor s of B
        Adjust the φ-function inputs in s
    for each child c of B in the dominator tree
        DVNT(c)
    Clean up the hash table after leaving this scope
```

Figure 4. Dominator-based value numbering technique

to an algorithm that extends value numbering to larger regions by using 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. Figure 4 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. This eliminates the need for the *name* array because each 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.,* $\langle x_0 \rangle$). When a redundant computation of an 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 canuse this replacement scheme over a limited region of the code.

1. The value number can replace a redundant computation in any block dominated by the first occurrence.
2. The value number can replace a redundant evaluation that is a parameter to a ϕ-node

corresponding to control flow from a block dominated by the first occurrence. To find these ϕ-node parameters, we compute the *dominance frontier* of the block containing the first occurrence of the expression. The *dominance frontier* of node X is the set of nodes Y such that X dominates a predecessor of Y, but X does not strictly dominate Y (*i.e.,* $\mathrm{DF}(X) = \{Y \mid \exists P \in \mathrm{Pred}(Y), X \geqq P \text{ and } X \not\gg Y\}$).*

In both cases, we know that control must flow through the block where the first evaluation occurred (defining the SSA name's value).

The ϕ-functions require special treatment. Before the compiler can analyze the ϕ-functions in a block, it must previously have assigned value numbers to all of the inputs. This is not possible in all cases; specifically, any ϕ-function input whose value flows along a back edge (with respect to the dominator tree) cannot have a value number. If any of the parameters of a ϕ-function have not been assigned a value number, then the compiler cannot analyze the ϕ-function, and it must assign a unique, new value number to the result. The following two conditions guarantee that all ϕ-function parameters in a block have been assigned value numbers:

1. When procedure DVNT (see Figure 4) 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.

If the above conditions are met, we can analyze the ϕ-functions in a block and decide if they can be eliminated. A ϕ-function can be eliminated if it is meaningless or redundant. A ϕ-function is *meaningless* if all its inputs have the same value number. A meaningless ϕ-function can be removed if the references to its result are replaced with the value number of its input parameters. A ϕ-function is *redundant* if it computes the same value as another ϕ-function in the same block. The compiler can identify redundant ϕ-functions 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-functions in different blocks.

After value numbering the ϕ-functions and instructions in a block, the algorithm visits each successor block and updates any ϕ-function inputs that come from the current block. This involves determining which ϕ-function 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-functions.

To illustrate how the algorithm works, we will apply it to the code fragment in Figure 5. The first block processed will be B_1. Since none of the expressions 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 $\langle v_0 \rangle$. Before we finish processing block B_2, we must fill in the ϕ-function parameters in its successor block, B_4. The first argument of ϕ-functions in B_4 corresponds to input from block B_2, so we replace u_0, x_0, and y_0 with $\langle u_0 \rangle$, $\langle v_0 \rangle$, and $\langle v_0 \rangle$, respectively.

* The SSA-construction algorithm uses dominance frontiers to place ϕ-nodes[6].

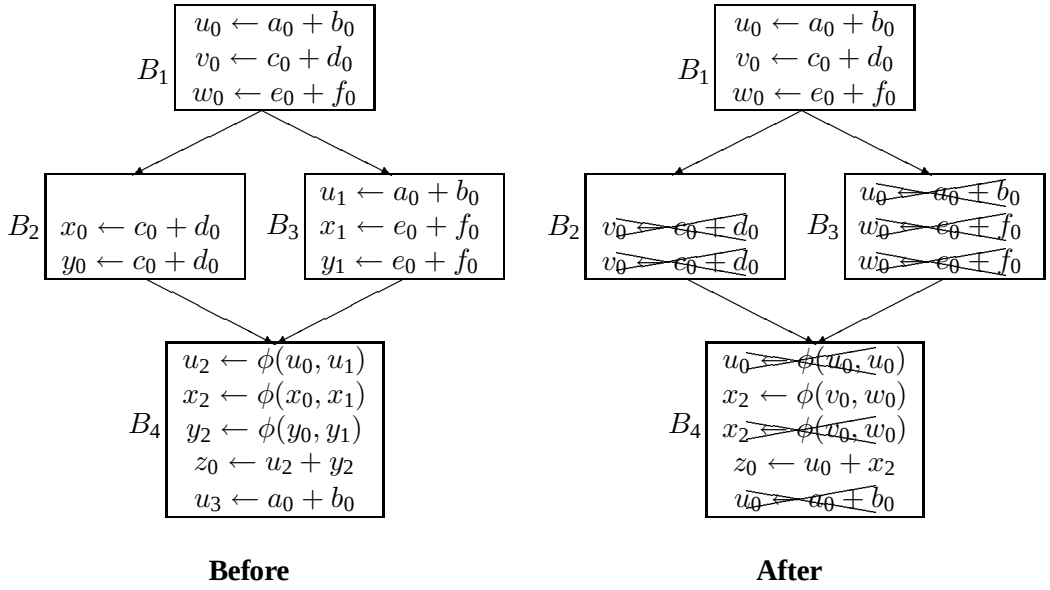

Figure 5. Dominator-tree value-numbering example

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 $\langle u_0 \rangle$, $\langle w_0 \rangle$, and $\langle w_0 \rangle$, respectively, and remove the assignments. To finish processing B_3, we fill in the second parameter of the ϕ-functions in B_4 with $\langle u_0 \rangle$, $\langle w_0 \rangle$, and $\langle w_0 \rangle$, respectively.

The final block processed will be B_4. The first step is to examine the ϕ-functions. Notice that we are able to examine the ϕ-functions 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-function defining u_2 is meaningless because all its parameters are equal (They have the same value number – $\langle u_0 \rangle$). Therefore, we eliminate the ϕ-function by assigning u_2 the value number $\langle u_0 \rangle$. It is interesting to note that this ϕ-function 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 that the ϕ-function for u was inserted during the construction of SSA form. The second ϕ-function combines the values v_0 and w_0. Since this is the first appearance of a ϕ-function with these parameters, x_2 is assigned its SSA name as its value number. The ϕ-function defining y_2 is redundant because it is equal to x_2. Therefore, we eliminate this ϕ-function by assigning y_2 the value number $\langle x_2 \rangle$. When processing the assignments in the block, we replace each operand by its value number. This results in the expression $\langle u_0 \rangle + \langle x_2 \rangle$ in the assignment to z_0. The assignment to u_3 is eliminated by giving u_3 the value number $\langle u_0 \rangle$.

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-based value numbering can remove the assignments to u_2, y_2, and u_3.

Incorporating value numbering into SSA construction

We have described dominator-based 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. The advantage of combining the steps is to improve the performance of the optimizer by reducing the amount of work performed and by reducing the size of the routine's SSA representation. The algorithm for dominator-based value numbering during SSA construction is presented in Figure 6. There is a great deal of similarity between the value numbering process and the renaming process during SSA construction [6]. 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. Notice that each element in the *VN* array in Figure 4 represents a value number, but the *VN* array in Figure 6 represents *stacks* of value numbers.
- Before inventing a new name for each ϕ-function or assignment, we first check if it can be eliminated. If so, we push the value number of the ϕ-function or assignment onto the stack for the defined name.

Unified hash table

A further improvement to hash-based value numbering is possible. We walk the dominator tree using a unified table (*i.e.,* a single hash table for the entire routine). Figure 7 illustrates how this technique differs from dominator-based value numbering. Since blocks B_2 and B_3 are siblings in the dominator tree, the entry for $a + b$ would be removed from the scoped hash table after processing block B_2 and before processing block B_3. Therefore, the two occurrences of the expression will be assigned different value numbers. On the other hand, no hash-table entries are removed when using a unified table. This allows both occurrences of $a + b$ to be assigned the same value number – $\langle x_0 \rangle$.

Using a unified hash-table has one important algorithmic consequence. Replacements cannot be performed on-line because the table no longer reflects availability. In previous algorithms, the existence of an expression in the hash table meant that the expression was computed earlier in the program. No such relationship between expressions exists under this approach. Thus, we cannot immediately remove expressions found in the table. In the example, it would be unsafe to remove the computation of $a + b$ from block B_3. Computations that are simplified, such as the meaningless ϕ-function in block B_4, may be directly removed. Thus, we must use a second pass over the code to eliminate redundancies. Fortunately, using the unified hash table results in the consistent naming scheme over the entire routine which is required by AVAIL-based removal and partial redundancy elimination (described later).

Strictly speaking, the unified hash table algorithm is not a global technique because it only works on acyclic subgraphs. In particular, it cannot analyze ϕ-functions in blocks with incoming back edges, and therefore it must assign a unique value number to any ϕ-function in such a block.

VALUE PARTITIONING

Alpern, Wegman, and Zadeck[2] presented a technique that uses a variation of Aho, Hopcroft, and Ullman's[12] formulation of Hopcroft's DFA-minimization algorithm to partition values into

```
procedure rename_and_value_number(Block B)
    Mark the beginning of a new scope
    for each φ-function p for name n in B
        if p is meaningless or redundant with value number v
            PUSH(p, VN[n])
            Remove p
        else
            Invent a new value number v for n
            PUSH(v, VN[n])
            Add p to the hash table
    for each assignment a of the form "x ← y op z" in B
        Overwrite y with TOP(VN[y]) and z with TOP(VN[z])
        expr ← ⟨y op z⟩
        if expr can be simplified to expr'
            Replace a with "x ← expr'"
            expr ← expr'
        if expr is found in the hash table with value number v
            PUSH(v, VN[x])
            Remove a
        else
            Invent a new value number v for x
            PUSH(v, VN[x])
            Add expr to the hash table with value number v
    for each successor s of B
        Adjust the φ-function 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 φ-function or assignment a in the original B
        for each name n defined by a
            POP(VN[n])
```

Figure 6. Value numbering during SSA construction

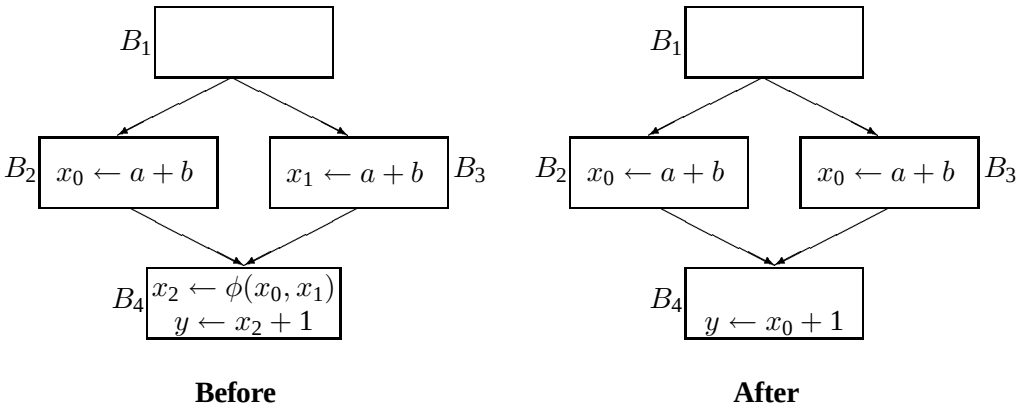

Figure 7. Unified hash table

congruence classes. It operates on the SSA form of the routine[6]. 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 SSA name in the routine to be congruent only to itself; however, the solution we seek is the *maximum fixed point* – the solution that contains the most congruent values.

The algorithm we use differs slightly from Alpern, Wegman, and Zadeck's. They describe an algorithm that operates on a structured programming language, where the SSA form is modified with ϕ_{if}-functions that represent `if-then-else` structures and ϕ_{enter} and ϕ_{exit}-functions that represent loop structures. These extensions to SSA form allow ϕ_{if}-functions to be compared to ϕ_{if}-functions in different blocks. The same is true for ϕ_{enter} and ϕ_{exit}-functions. In order to be more general, our implementation of value partitioning operates on pure SSA form, which means that ϕ-functions in different blocks cannot be congruent.

Figure 8 shows the partitioning algorithm. 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-functions in the routine are renumbered based on the congruence classes so that all congruent definitions have the same name. In other words, for each SSA name, n, we replace each occurrence of n in the program with the name chosen to represent the congruence class containing n. Because the effects of partitioning and renumbering are similar to those of value numbering using the unified hash table described in the previous section, we think of this technique as a form of global (or intraprocedural) value numbering.* Value partitioning and the unified hash table algorithm do not necessarily discover the same equivalences, but they both provide a consistent naming of the expressions throughout the entire routine.

Partitioning and renaming alone will not improve the running time of the routine; we must

* Rosen, Wegman, and Zadeck[13] describe a technique called *global value numbering*. It is an interesting and powerful approach to redundancy elimination, but it should not be confused with value partitioning.

```
Place all values computed by the same opcode in the same
    congruence classes
worklist ← the classes in the initial partition
while worklist ≠ ∅
    Select and delete an arbitrary class c from worklist
    for each position p of a use of x ∈ c
        touched ← ∅
        for each x ∈ c
            Add all uses of x in position p to touched
        for each class s such that ∅ ⊂ (s ∩ touched) ⊂ s
            Create a new class n ← s ∩ touched
            s ← s − n
            if s ∈ worklist
                Add n to worklist
            else
                Add smaller of n and s to worklist
```

Figure 8. Partitioning algorithm

also find and remove the redundant computations. We explore three possible approaches: dominator-based removal, AVAIL-based removal, and partial redundancy elimination.

Dominator-based removal

Alpern, Wegman, and Zadeck[2] suggest removing any computation that is dominated by a definition from the same congruence class. In Figure 9, the computation of z is a redundancy that this method can eliminate. Since the computation of z in block B_1 dominates the

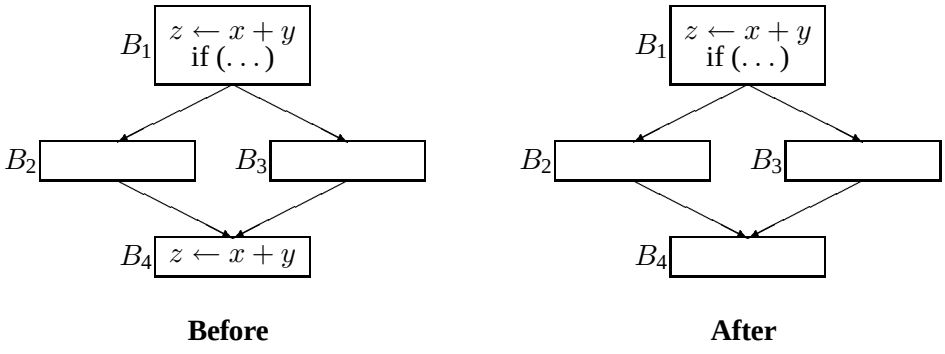

Figure 9. Program improved by dominator-based removal

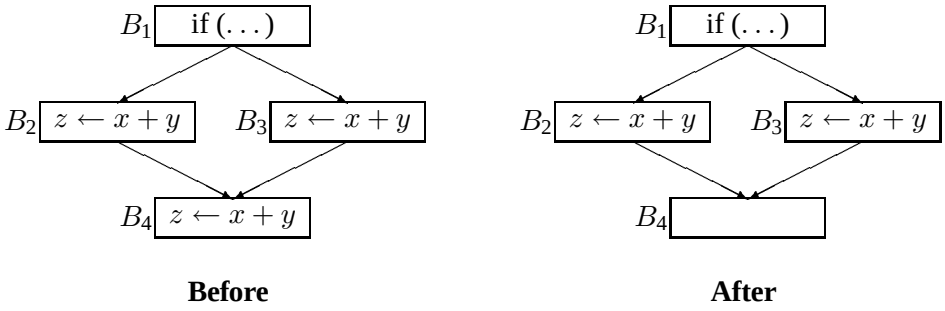

Figure 10. Program improved by AVAIL-based removal

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.

AVAIL-based removal

The classic approach to redundancy elimination is to remove computations that are in the set of available expressions (AVAIL)[3] at the point where they appear in the routine. 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 z in Figure 9 will be removed because it is in the AVAIL set. In fact, any computation that 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 10, z 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 z from B_4.

Properties of the value numbering and renaming algorithm let us simplify the formulation of AVAIL. The traditional data-flow equations deal with lexical *names* while our equations deal with *values*. This is an important distinction. 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 11. If this code is produced by value numbering and renaming, the two assignments to Z must be equal. 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

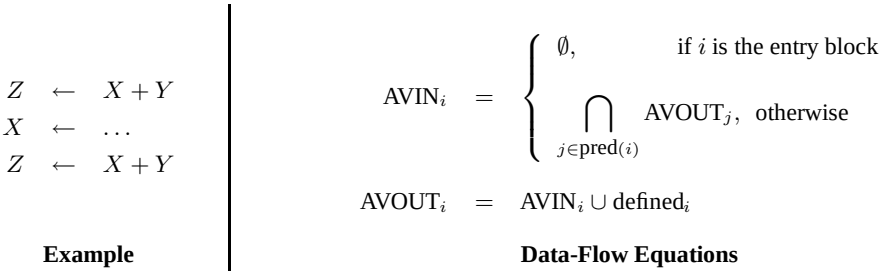

Figure 11. AVAIL-based removal

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 11.

Partial redundancy elimination

Partial redundancy elimination (PRE) is an optimization introduced by Morel and Renvoise[4]. Partially redundant computations are redundant along some, but not all, execution paths. PRE operates by discovering partially redundant computations, inserting code to make many of them fully redundant,* and then removing all redundant computations.

In Figures 9 and 10, the computations of z 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 in Figure 12

* PRE only inserts a copy of an evaluation if it can prove that the insertion, followed by removal of the newly redundant code, makes no path longer. In practice, this prevents it from removing some partially redundant expressions inside if-then-else constructs.

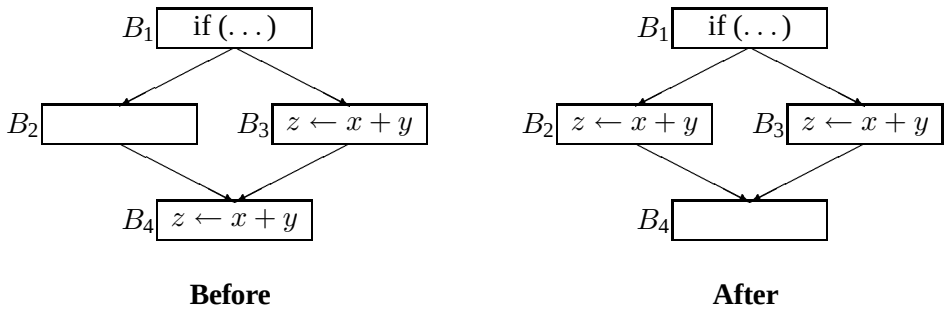

Figure 12. Program improved by partial redundancy elimination

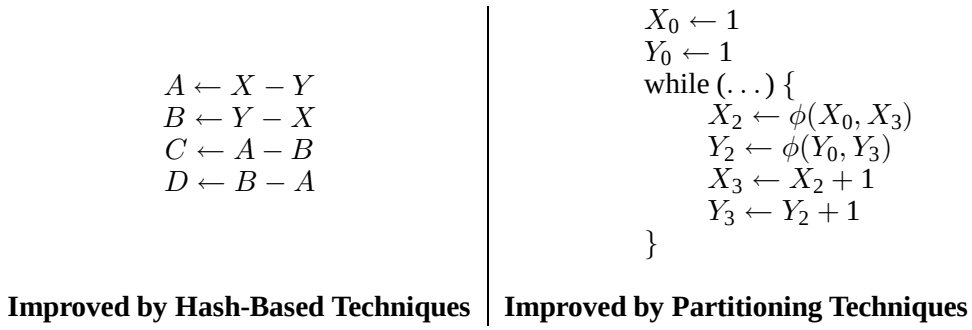

Figure 13. Comparing the techniques

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. It inserts a copy of the computation in B_2, making the computation in B_4 redundant. Next, it removes the computation from B_4. 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.

COMPARING THE TECHNIQUES

While the effects of hash-based value numbering using the unified table and value partitioning are similar, the two techniques can discover different sets of equivalences. Assume that X and Y are known to be equal in the code fragment in the left column of Figure 13. 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. The hash-based approach combines congruence finding and constant propagation to produce an optimization that is more powerful than the sum of its parts. Click described precisely when combining optimizations is profitable[14].

Now consider the code fragment in the right column of Figure 13. 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

* Two expressions can only be congruent if they have the same operator. Thus, the partitioning technique cannot discover that `1+1` and `2*1` compute the same value.

these techniques have visited X_3 or Y_3. Therefore, they must assign different value numbers 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 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 proceed to prove as many equalities as possible.

We should point out that eliminating more redundancies does not necessarily result in reduced execution time. This effect is a result of the way different optimizations interact. The primary interactions are with register allocation and with optimizations that combine instructions, such as constant folding or peephole optimization. Each replacement affects register allocation because it has the potential of shortening the live ranges of its operands and lengthening the live range of its result. Because the precise impact of a replacement on the lifetimes of values depends completely on context, the impact on demand for registers is difficult to assess. In a three-address intermediate code, each replacement has two opportunities to shorten a live range and one opportunity to extend a live range. We believe that the impact of replacements on the demand for registers is negligible; however, this issue deserves more study.

The interaction between value numbering and other optimizations can also affect the execution time of the optimized program. The example in Figure 14 illustrates how removing more redundancies may not result in improved execution time. The code in block B_1 loads the value of the second element of a common block called "foo", and the code in block B_2 loads the first element of the same common block. Compared to value numbering over single basic blocks, value numbering over extended basic blocks will remove more redundancies. In particular, the computation of register r_5 is not needed because the same value is in register r_1. However, the definition of r_1 is no longer used in block B_1 due to the constant folding in the definition of r_3. The definition of r_1 is now partially dead because it is used along the path through block B_2 but not along the path through B_3. If the path through block B_3 is taken at run time, the computation of r_1 will be unused. On the other hand, value numbering over single basic blocks did not remove the definition of r_5, and the definition of r_1 can be removed by dead code elimination. The result is that both paths through the CFG are as short as possible. Other optimizations that fold or combine optimizations, such as constant propagation or peephole optimization, can produce analogous results. In our test suite, the `saturr` routine exhibits this behavior.

EXPERIMENTAL RESULTS

Even though we can prove that each of the three partitioning techniques and each form of hash-based value numbering is never worse than its predecessor in terms of eliminating redundancies, 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. The compiler is centered around our intermediate language, called ILOC (pronounced "eye-lock"). ILOC is a pseudo-assembly language for a RISC machine with an arbitrary number of symbolic registers. LOAD and STORE operations are provided to access memory, and all computations operate on symbolic registers. The front end translates Fortran into ILOC. The optimizer is organized as a collection of Unix filters that consume and produce ILOC. This design allows us to easily apply optimizations in almost any order. The back end produces code instrumented to count the number of ILOC

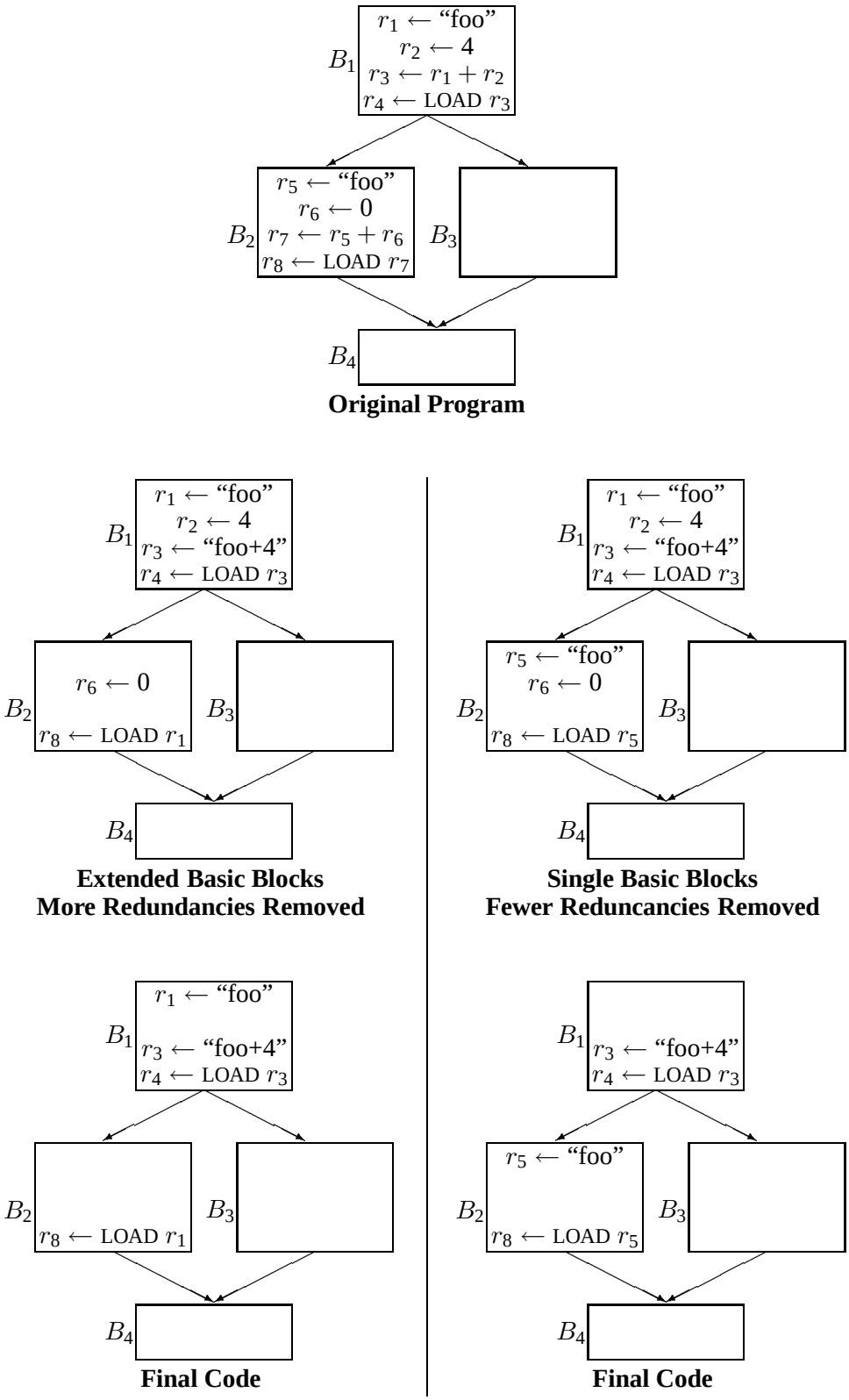

Figure 14. Interaction with other optimizations

Figure 15. Comparison of hash-based techniques – SPEC benchmark

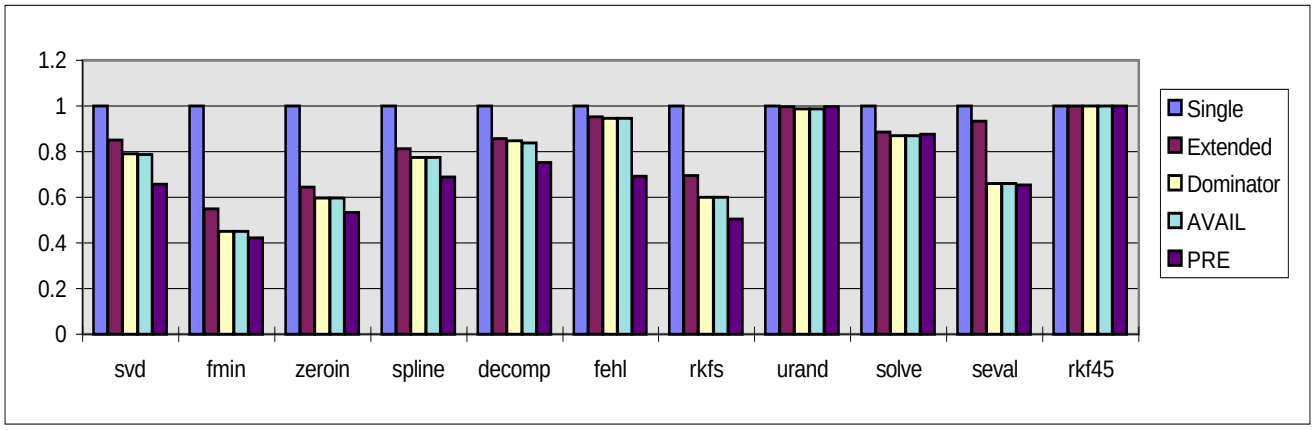

Figure 16. Comparison of hash-based techniques – FMM benchmark

instructions executed.

Comparisons were made using routines from a suite of benchmarks consisting of routines drawn from the SPEC benchmark suite[15] and from Forsythe, Malcolm, and Moler's book on numerical methods[16]. We refer to the latter as the FMM benchmark. Each routine was optimized in several different ways by varying the type of redundancy elimination (value numbering followed by code removal or motion).* To achieve accurate comparisons, we varied only the type of redundancy elimination performed. The complete results are shown in Figures 15 through 20. Each bar represents dynamic counts of ILOC operations, normalized against the the leftmost bar. Routines are optimized using the sequence of global reassociation[17], redundancy elimination, global constant propagation[18], global peephole optimization, dead code elimination[6], operator strength reduction[19, 20], redundancy elimination, global constant propagation, global peephole optimization, dead code elimination, 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 hash-based approaches use the unified table method. Its global name space is needed for either AVAIL-based removal or PRE. All tests were run on a two-processor Sparc10 model 512 running at 50 MHz with 1 MB cache and 115 MB of memory.

Figures 15 and 16 compare the hash-based techniques. In general, each refinement to the technique results in an improvement in the results. We see significant improvements when moving from single basic blocks to extended basic blocks and again to dominator-based removal. One surprising aspect of this study is that the differences between dominator-based removal and AVAIL-based removal are small in practice.† The differences between AVAIL-based removal and PRE are significant. The ability of PRE to move invariant code out of loops contributes greatly to this improvement. However, there are some examples where our value-based formulation of AVAIL-based removal is better than PRE, which operates on lexical names. Figures 17 and 18 compare the partitioning techniques. The results are similar to the

* The sizes of the test cases for `matrix300` and `tomcatv` have been reduced to ease testing.

† This suggests that either (1) the situation depicted in Figure 10 occurs infrequently in the tested codes, or (2) some combination of the other optimizations catch this situation. It appears that the first explanation is the correct one.

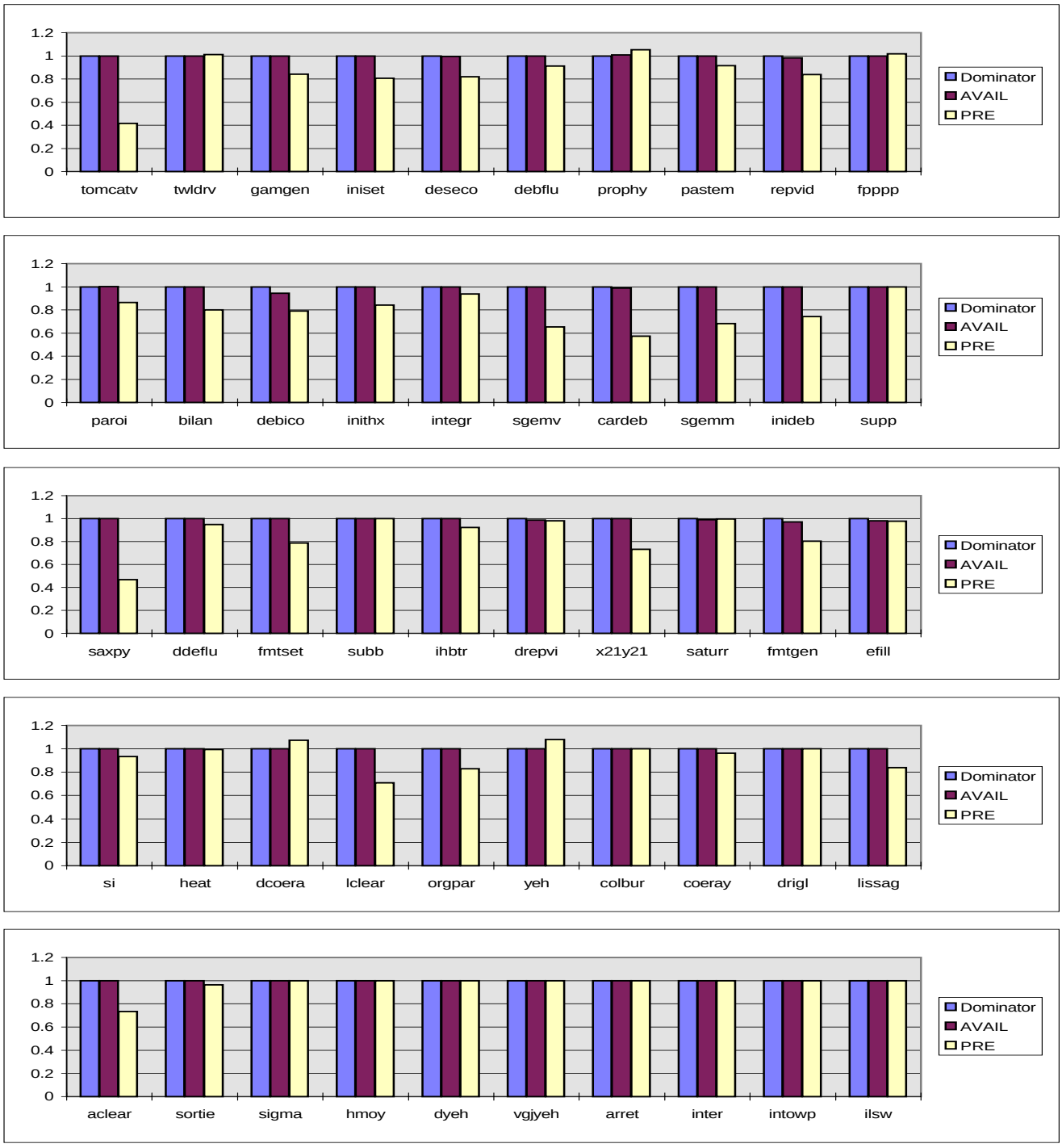

Figure 17. Comparison of partitioning techniques – SPEC benchmark

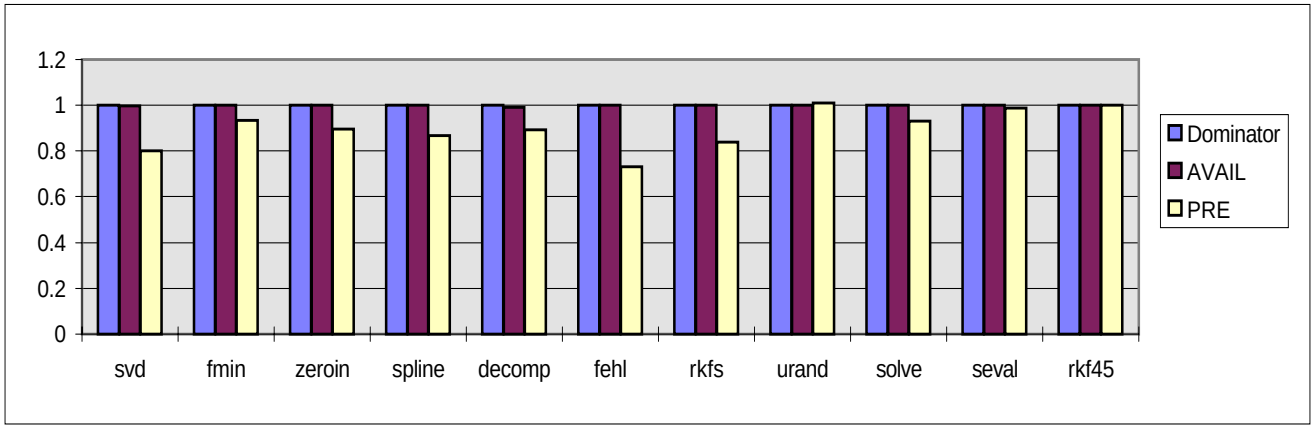

Figure 18. Comparison of partitioning techniques – FMM benchmark

results from the hash-based comparison.

Figures 19 and 20 compare the unified hash-table version of dominator-tree value numbering with value partitioning under each of the code removal and motion strategies. Hash-based value numbering almost always eliminates significantly more redundancies than value partitioning. This is due to the fact that hash-based value numbering can fold constants and simplify algebraic identities. These are more frequent in practice than the global redundancies identified by value partitioning.

Table I compares the time required by hash-based value numbering and value partitioning for some of the larger routines in the test suite. The number of blocks, SSA names, and operations are given to indicate the size of the routine being optimized. In all cases, hash-based value numbering runs faster than value partitioning.

routine	*blocks*	*SSA names*	*operations*	*hash-based*	*partitioning*
`tomcatv`	131	3366	3222	0.05	0.07
`ddeflu`	109	9034	6687	0.11	0.81
`debflu`	116	7183	4320	0.08	0.93
`deseco`	251	16521	12932	0.30	1.85
`twldrv`	261	26948	14298	0.40	6.09
`fpppp`	2	26590	25934	0.63	1.16

Table I. Compile times of value numbering techniques

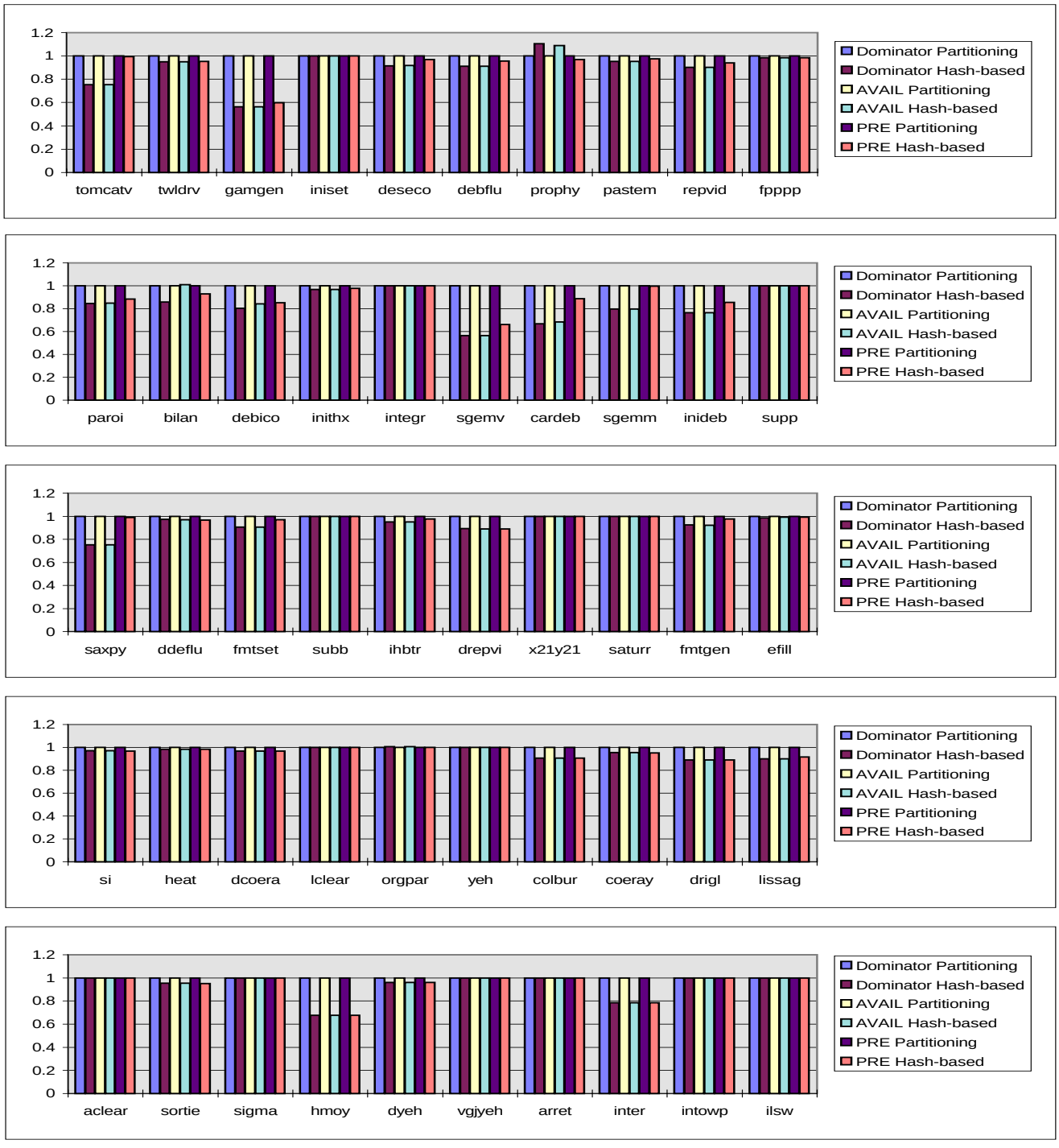

Figure 19. Comparison of hash-based vs. partitioning techniques – SPEC benchmark

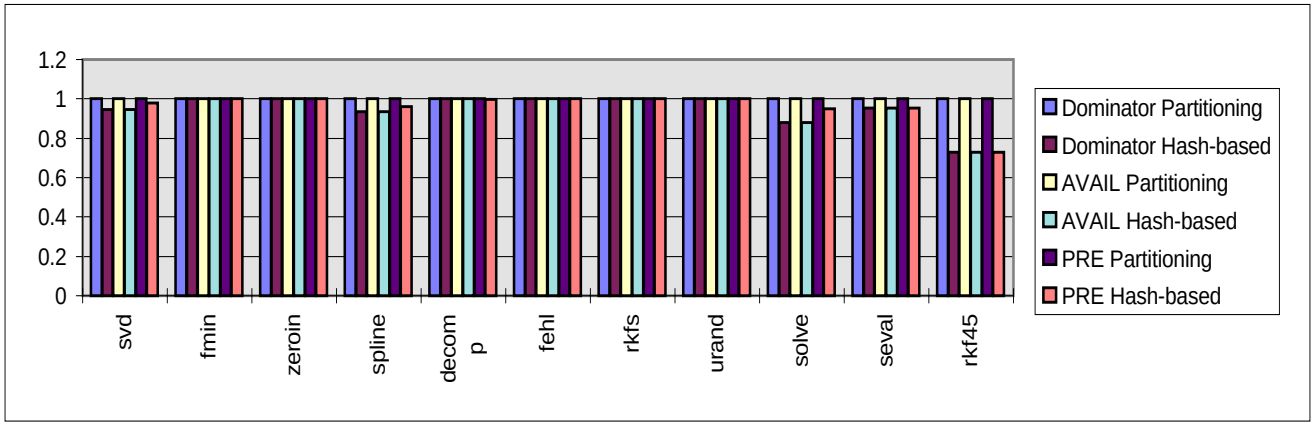

Figure 20. Comparison of hash-based vs. partitioning techniques – FMM benchmark

SUMMARY

In this paper, we study a variety of redundancy elimination techniques. We have introduced a technique for applying hash-based value numbering over a routine's dominator tree. This technique is superior in practice with the value partitioning techniques, while being faster and simpler. Additionally, we have improved the effectiveness of value partitioning 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.

ACKNOWLEDGEMENTS

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. The referees made a number of detailed comments and suggestions that improved both the exposition and content of the paper.

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, Nat McIntosh, 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. John Cocke and Jacob T. Schwartz, 'Programming languages and their compilers: Preliminary notes', *Technical report*, Courant Institute of Mathematical Sciences, New York University, 1970.

2. Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck, 'Detecting equality of variables in programs', *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1988, pp. 1–11.
3. John Cocke, 'Global common subexpression elimination', *SIGPLAN Notices*, **5**(7), 20–24 (1970). *Proceedings of a Symposium on Compiler Optimization.*
4. Etienne Morel and Claude Renvoise, 'Global optimization by suppression of partial redundancies', *Communications of the ACM*, **22**(2), 96–103 (1979).
5. Jiazhen Cai and Robert Paige, "'Look Ma, no hashing, and no arrays neither"', *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, Orlando, Florida, January 1991, pp. 143–154.
6. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck, 'Efficiently computing static single assignment form and the control dependence graph', *ACM Transactions on Programming Languages and Systems*, **13**(4), 451–490 (1991).
7. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.
8. Charles N. Fischer and Jr. Richard J. LeBlanc, *Crafting a Compiler with C*, The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1991.
9. John H. Reif, 'Symbolic programming analysis in almost linear time', *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, Tucson, Arizona, January 1978, pp. 76–83.
10. Thomas Lengauer and Robert Endre Tarjan, 'A fast algorithm for finding dominators in a flowgraph', *ACM Transactions on Programming Languages and Systems*, **1**(1), 121–141 (1979).
11. Matthew S. Hecht, *Flow Analysis of Computer Programs*, Programming Languages Series, Elsevier North-Holland, Inc., 52 Vanderbilt Avenue, New York, NY 10017, 1977.
12. Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Massachusetts, 1974.
13. Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck, 'Global value numbers and redundant computations', *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1988, pp. 12–27.
14. Cliff Click, 'Combining analyses, combining optimizations', *Ph.D. Thesis*, Rice University, 1995.
15. SPEC release 1.2, September 1990. Standards Performance Evaluation Corporation.
16. George E. Forsythe, Michael A. Malcolm, and Cleve B. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, Englewood Cliffs, New Jersey, 1977.
17. Preston Briggs and Keith D. Cooper, 'Effective partial redundancy elimination', *SIGPLAN Notices*, **29**(6), 159–170 (1994). *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*.
18. Mark N. Wegman and F. Kenneth Zadeck, 'Constant propagation with conditional branches', *ACM Transactions on Programming Languages and Systems*, **13**(2), 181–210 (1991).
19. Frances E. Allen, John Cocke, and Ken Kennedy, 'Reduction of operator strength', in Steven S. Muchnick and Neil D. Jones (eds.), *Program Flow Analysis: Theory and Applications*, Prentice-Hall, 1981.
20. Keith D. Cooper, L. Taylor Simpson, and Christopher A. Vick, 'Operator strength reduction', *Technical Report CRPC-TR95635-S*, Center for Research on Parallel Computation, Rice University, October 1995.