RICE UNIVERSITY

Value-Driven Redundancy Elimination

by

Loren Taylor Simpson

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Keith D. Cooper, Associate Professor, Chair
Department of Computer Science
Rice University

Linda Torczon, Faculty Fellow
Department of Computer Science
Rice University

Ken Kennedy, Noah Harding Professor
Department of Computer Science
Rice University

Sarita Adve, Assistant Professor
Department of Electrical and Computer Engineering
Rice University

Houston, Texas

April, 1996

Value-Driven Redundancy Elimination

Loren Taylor Simpson

Abstract

Value-driven redundancy elimination is a combination of value numbering and code motion. Value numbering is an optimization that assigns numbers to values in such a way that two values are assigned the same number 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. Code motion is an optimization that attempts to move instructions to less frequently executed locations. Traditional techniques must assume that every definition produces a distinct value. Therefore, an instruction cannot move past a definition of one of its subexpressions. This restriction can be relaxed when certain definitions are known to produce redundant values. By understanding how these two optimizations interact, we can simplify each of them, and the resulting combination will be more powerful than the sum of the two parts. Value numbering will be simpler because it need not be concerned with eliminating instructions, and code motion will be simpler because identifying subexpressions is not necessary.

This research investigates this value-driven approach to redundancy elimination. We improve upon the known algorithms for both value numbering and code motion, and we show experimental evidence of their effectiveness.

Acknowledgments

To say that this thesis has been an individual effort would be grossly misleading. Many people have contributed to this work in tangible and intangible ways. I would first like to thank God for leading me to Rice and surrounding me with wonderful friends and coworkers. My wife, Sallye, and my parents and her parents have provided me with constant support and encouragement.

The members of my committee are Keith Cooper, Linda Torczon, Ken Kennedy, and Sarita Adve. They have contributed greatly to this research. I admire and respect them deeply, and I am honored to be associated with them.

This research has been part of the Massively Scalar Compiler Project at Rice University. The project is supported by ARPA, and Vivek Sarkar of IBM has provided my support. The members of the group (past and present) are Keith Cooper, Linda Torczon, Ken Kennedy, Preston Briggs, Tim Harvey, Lisa Thomas, Rob Shillingsburg, Cliff Click, Chris Vick, John Lu, Edmar Wienskoski, Phil Schielke, and Linlong Jiang. Preston Briggs was a great influence during my early years as a graduate student, and I regret that he is no longer a member of our project. Tim Harvey has been the kind of friend who will go out of his way to help someone, and then deny that it was any trouble at all.

Bob Morgan of DEC has shown a great deal of interest in the project, and he has encouraged our research in the area of redundancy elimination. Dave Spott and David Wallace of Sun Microsystems are implementing the algorithm for SCC-based value numbering in their next generation compiler.

My time as a graduate student at Rice has been one of the most rewarding periods in my life. This is due in part to the standard of excellence set by the past and present Rice graduate students.

Contents

Illustrations

Chapter 1

Introduction

A compiler is a program that translates programs in one language (called the *source language*) to equivalent programs in another language (called the *target language*). Usually, the source language is a high-level language written by a programmer and the target language is the machine code for some computer. The primary goals of the compiler are to:

1. Preserve the meaning of the original code
2. Produce efficient code
3. Compile quickly

Figure 1.1 shows the organization of a typical compiler consisting of three stages. The *front end* must perform scanning, parsing, and context-sensitive analysis. The *optimizer* transforms the program to improve its efficiency. Generally, the optimizer is organized as a sequence of passes – each with a specific purpose. The *back end* generates the target language. It must perform register allocation, scheduling, and instruction selection. There are many interesting problems involved in building both the front end and the back end of a compiler, but this thesis is concerned primarily with the compiler's optimizer.

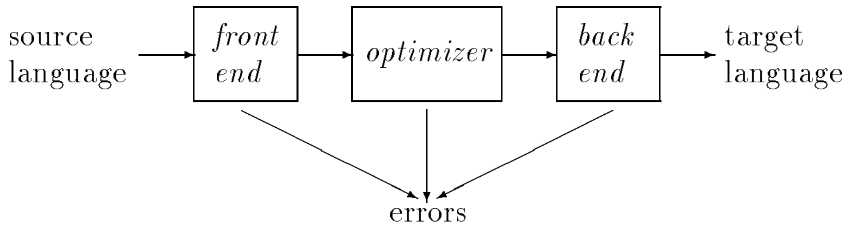


Figure 1.1 Organization of a typical compiler

1.1 Intermediate Representations

The compiler transforms a program represented in the source language to an equivalent program represented in the target language. Along the way, a variety of *intermediate representations* will be used. The design of the intermediate representations is a fundamental part of the compiler development process. The intermediate representation allows the three stages to communicate and it determines the amount of information about the program that is available.

Often, the compiler will build an *abstract syntax tree* (AST) during parsing. It is very similar to a parse tree except that many of the unnecessary nodes are removed. The AST is very convenient for performing context-sensitive analysis and certain high-level optimizations.

The compiler can generate *three-address code* during a walk of the AST. Three-address code is a sequence of instructions of the form:

$$x \leftarrow y \text{ op } z$$

Three-address code gets its name because each instruction can access three names. During the generation of three-address code, the compiler may create temporary names for results that are not given a name by the programmer. For example, a large expression will be calculated by a sequence of three-address instructions, and each instruction will define a compiler-generated temporary name. When the value of the entire expression has been calculated, it can be stored in one of the program's variables.

The three-address code can be divided into *basic blocks*, or maximal sequences of straight line code. A basic block has the property that if one instruction executes, they all execute in order. The basic blocks are organized into a *control-flow graph* (CFG). The nodes in the CFG represent basic blocks and the edges correspond to possible control flow from one basic block directly to another.

1.2 Static Single Assignment Form

Many of the optimizations presented here are based on static single assignment (SSA) form [18]. 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. A complication arises in routines with more than one basic block. Values can flow into a block from more than one definition site, and each site has a unique SSA name for the item. Consider the example in

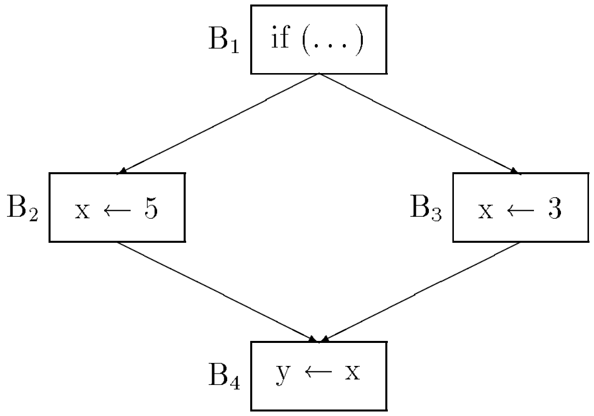


Figure 1.2 Program before conversion to SSA

Figure 1.2. There are two definition points for the name x referenced in B_4. If each definition point is overwritten with a unique SSA name, how can we correctly replace the reference to x? Special assignments called ϕ-functions are used to solve the problem. These ϕ-functions are placed at the routine's join points (that is, at the beginning of the basic blocks which have more than one predecessor). One ϕ-node is inserted at each join point for each name in the original routine. This name is called the ϕ-node's original name. 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* of each value – the earliest location where the joined value exists [39].

The ϕ-functions provide a single definition for a name that had more than one definition (on different control-flow paths) in the original routine. Each ϕ-node 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 item in the basic block are replaced by the new name. The number of inputs for a ϕ-node is equal to the number of predecessors of the block in which the ϕ-node appears. The semantics of the ϕ-function are simple. The ϕ-node selects the value of the input that corresponds to the block from which control is transferred and assigns this value to the result. When ϕ-functions are properly inserted, it becomes possible for any routine to be renamed to produce an equivalent routine in which each name has exactly one definition point. The SSA form of the example in Figure 1.2 is shown in Figure 1.3. Renaming has already been performed. If control is transferred from block B_2 to B_4,

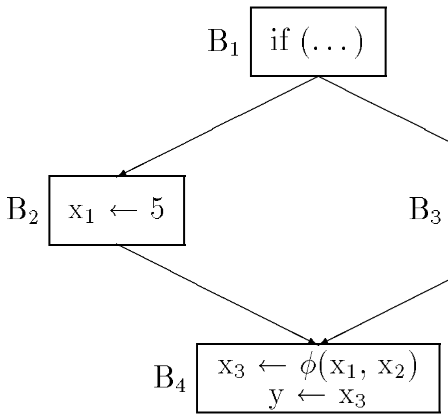


Figure 1.3 Program after conversion to SSA

the ϕ-node will assign x_3 the value from x_1. If control is transferred from block B_3 to B_4, the ϕ-node will assign x_3 the value from x_2.

1.3 Optimization

This thesis focuses on the compiler's optimizer. Actually, the term *optimization* is somewhat misleading. It implies that the code produced is optimal. In reality, the optimizer transforms the intermediate representation in a way that is believed to improve the program. Typically, improvements focus on the execution speed of the program. However, other improvements, such as reducing memory requirements, are also possible. There is no guarantee that the resulting code cannot be improved. In fact, it is possible that the optimizer will make the program worse. Therefore, many tradeoffs must be considered when designing an optimizer. The fundamental responsibility of the optimizer is to preserve *observational equivalence* [37]. In other words, the optimized program must produce the same results as the unoptimized program for all possible inputs.

A typical optimizer is organized as a sequence of passes (or *optimizations*). Each pass tries to either improve the running time of the program or decrease its space requirements. Some passes (or all passes) may be repeated. Each optimization performs a specific task. Examples include:

- discover and propagate a constant value
- move a computation to a less frequently executed place

- specialize a computation based on context
- discover a redundant computation and remove it
- remove code that is useless or unreachable
- combine several instructions into some powerful instruction

The order in which the transformations are performed is important for several reasons. One optimization may require code in a certain "shape". One optimization may discover "facts" that another needs. One optimization may introduce opportunities or destroy opportunities for another optimization [38].

1.4 Classification of Optimizations

When discussing a particular optimization, it is often helpful to compare it to other optimizations. To do this effectively, we must understand which optimizations are good candidates for comparison. The first classification is based on the assumptions made about the target machine.

Machine independent Assumes no knowledge of the target machine

Machine dependent Uses a specific feature of the target machine

There are very few optimizations that are truly independent of the target machine. For example, removing a redundant computation would improve the execution time of the program on most machines. However, this change may increase the register pressure beyond what the target machine can support. If so, the register allocator will be forced to insert spill code that may be more expensive than the original computation. As a rule of thumb, machine independent optimizations apply to a broad class of machines and they ignore many of the constraints present in real machines. Examples include removing redundant computations and moving instructions to less frequently executed locations. Further, optimizations whose primary focus is to take advantage of some feature of a particular architecture are called machine dependent. Examples include rewriting memory operations to take advantage of specific addressing modes and reordering instructions to reduce the number of pipeline stalls.

The second classification is based on the *scope* of the optimization – how much of the program must be analyzed.

Local Analyze and transform a single basic block

Global Analyze and transform a single procedure

Interprocedural Analyze and transform the whole program

As we shall see, there are several scopes that fall between these major levels. For example, in Chapter 2, we will discuss *extended basic blocks*, which lie between local and global.

1.5 Safety, Opportunity, and Profitability

The primary criteria for evaluating optimizations are *safety*, *opportunity*, and *profitability*. Safety deals with the issue of whether or not an optimization will destroy the observational equivalence property. An unsafe optimization has the potential to change the observable behavior of a program. Unsafe optimizations should never be applied.

Opportunity is concerned with the amount of effort involved in identifying locations within the program where the transformation can be applied. The intermediate representation plays a significant role in this aspect of optimization. For example, locating the loops is rather simple using an abstract syntax tree, but it is more difficult if the program is represented by a control-flow graph. On the other hand, the programmer may have created a loop using GOTO's, which will not be visible in the abstract syntax tree. Another common way to identify opportunities is *data-flow analysis* – compile-time reasoning about the run-time flow of values.

Profitability deals with the amount of improvement expected from applying a transformation. The profitability of an optimization can be computed in a number of ways. Some transformations, such as *dead code elimination* – removing computations whose values are never used, are always profitable. Other transformations, such as removing redundant computations, are simply assumed to be profitable. We saw how this assumption can be violated in the previous section. The profitability of an optimization might be calculated at compile-time or at run-time. If the calculation shows that the transformation is profitable, then it will be applied. When the calculation is performed at run-time, the compiler must generate code with and without applying the transformation. Often, the compiler writer must consider the tradeoffs between opportunity and profitability when deciding which optimizations to implement. For example, a transformation with large amounts of compile time required to find opportunities and relatively small payoffs in profitability would not be very attractive. On the other hand, a transformation where opportunities are easy to find and profitability is large would be very attractive.

1.6 Redundancy Elimination

This research focuses on new techniques for compiler-based redundancy elimination. Optimizing compilers often attempt to eliminate redundant computations either by removing instructions or by moving instructions to less frequently executed locations. Historically, the algorithms aimed at removing instructions have been designed independently from those aimed at moving instructions. Usually, there is one optimization pass that attempts to determine when two instructions compute the same value and then decides if one of the instructions can be eliminated. A second optimization pass determines a set of locations where an instruction would compute the same result, and it selects the one that is expected to be the least frequently executed. This approach suffers from a phase ordering problem as well as the problem that no information is shared between the two passes. We believe the correct approach to redundancy elimination is a single optimization with two steps:

1. Determine which computations in the program compute the same value, and identify the *values* computed in the routine. We will refer to this step as *value numbering*, because we assign numbers to values so that two values have the same number if the compiler can prove they are equal.

2. Use the value numbers to remove instructions or move them to less frequently executed locations.

One advantage of formulating the problem in this manner is that it provides a good separation of concerns. Step 1 encodes the knowledge that it discovers into the choice of specific value numbers. Step 2 can rely on the numbers (encoded in the name space or in a set of tables) as a basis for its reasoning. Thus, we can select the algorithm for step one independently of the selection of the algorithm for step two. This research has improved upon the best known solution to each of these steps.

1.7 Organization of the Thesis

The remainder of this thesis is organized as follows: Chapter 2 will discuss the hash-based approach to value numbering. Chapter 3 will present a global algorithm for value numbering that is based on partitioning. Chapter 4 will present an original algorithm that combines the advantages of hash-based value numbering and value partitioning. Chapter 5 will explain techniques for removing instructions from a routine, and Chapter 6 will present techniques for moving instructions to less frequently executed locations. Chapter 7 will present a new algorithm that improves upon previous code motion frameworks by taking advantage of the results of value numbering. Chapter 8 will discuss a new technique for applying code motion techniques to relieve register pressure and to improve register allocation. All of the techniques described in this thesis have been implemented using the compiler developed by the Massively Scalar Compiler Project at Rice University. An experimental comparison of the techniques will be presented in Chapter 9. Chapter 10 will summarize the contributions and results of this thesis. We present an algorithm for operator strength reduction in Appendix A because the algorithm relies on redundancy elimination and it is centered around finding the strongly connected components of the SSA graph, just like the value numbering algorithm in Chapter 4.

Chapter 2

Hash-Based Value Numbering

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.

1. It assigns an identifying number (a *value number*) to each value computed by 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.

2. 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.

3. It uses value numbers to find redundant computations and remove them.

4. It discovers constant values, evaluates expressions whose operands are constants, and propagates them through the code.

Cocke and Schwartz describe a local technique that uses hashing to discover redundant computations and fold constants [17]. We believe the technique was originally invented by Balke at Computer Sciences Corporation [24]. Each unique value 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."

The algorithm is relatively simple. In practice, it is very fast. For each instruction from top to bottom in the block, it hashes the operator and the value numbers of the operands to obtain the unique value number that corresponds to the computation's value. If it has already been computed in the block, the expression 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 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 2.1. 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 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. Notice that a block can be a member of more than one extended basic block. For example, consider an if-then-else construct like the one shown in Figure 2.2 where block B_1 is contained in two extended basic blocks: B_1, B_2 and B_1, B_3. Value numbering over extended blocks works precisely because each value that flows into a block must flow from its predecessor and nowhere else. The blocks in the sequence can be processed by initializing their hash tables with the results of processing the previous block. In general, the extended basic blocks in a flow graph form a forest, which 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

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

Figure 2.1 Value numbering example

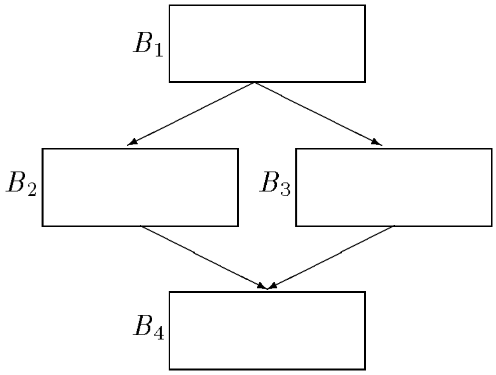


Figure 2.2 CFG for if-then-else construct

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.

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 [18]. The critical property of SSA that we require 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 2.1 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

The concept of dominance is very important in program optimization and the construction of SSA form. In a flow graph, if node X appears on every path from the start node to node Y, then X *dominates* Y ($X \geqq Y$) [33]. If $X \geqq Y$ 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 [25]. 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-node at the start of B or it flows through B's parent in the dominator tree. Notice that, for extended basic blocks, B_{i-1} is the immediate dominator of B_i, for $2 \leq i \leq n$. These observations led us to an algorithm for value numbering over the dominator tree. Bob Morgan of DEC also made these observations and encouraged us to pursue this approach.

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 2.3 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 [18].[1] 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. Before the compiler can analyze the ϕ-nodes in a block, it must have previously assigned value numbers to all of the inputs. This is not possible in all cases; specifically, any ϕ-node input whose value flows through a back edge cannot have a value number. If any of the parameters of a ϕ-node have not been assigned a value number, then the compiler cannot analyze the ϕ-node, and it must assign a unique new value number to the result. The following

[1] 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.*, $\mathrm{DF}(X) = \{Y \mid \exists P \in \mathrm{Pred}(Y), X \geqq P \text{ and } X \not\gg Y\}$).

```
procedure DVNT(Block b)
    Mark the beginning of a new scope
    for each φ-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] ← n
            Add p to the hash table
    for each assignment a of the form n ← 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] ← n
            Add exp to the hash table
    for each successor s of b
        Adjust the φ-node 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 2.3 Dominator-tree value numbering two conditions guarantee that all ϕ-node parameters in a block have been assigned value numbers.

1. When DVNT 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.

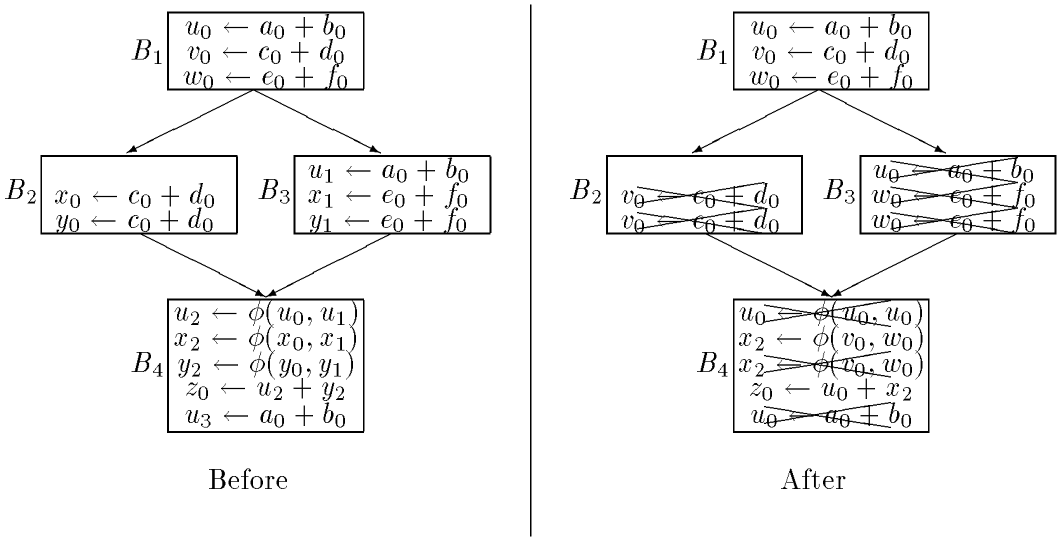
2. The block must have no incoming back edges.

If the above conditions are met, we can analyze the ϕ-nodes in a block and decide if they can be eliminated. 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.

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 2.4. 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.



u_0	u_0
v_0	v_0
w_0	w_0
x_0	v_0
y_0	v_0
u_1	u_0
x_1	w_0
y_1	w_0
u_2	u_0
x_2	x_2
y_2	x_2
z_0	z_0
u_3	u_0

Value Numbers

Figure 2.4 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 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.

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 (they have the same value number). 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 [18, 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.

The algorithm for dominator-tree value numbering during SSA construction is presented in Figure 2.5.

2.4 Unified Hash Table

A further improvement to dominator-tree 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) and replace each SSA name with its value number. Figure 2.6 illustrates how this technique differs from dominator-tree value numbering. Since blocks B and C are siblings in the dominator tree, the entry for $a + b$ would be removed from the scoped hash table after processing block B and before processing block C. 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.

Using a unified hash-table has a very important algorithmic consequence. Replacements cannot be performed on-line because the table no longer reflects availability. 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 C. However, computations that are simplified (such as the meaningless ϕ-node in block D) may be removed. Since we cannot remove all redundancies during value numbering, we must use a second pass over the code to perform replacement; we can use any of the techniques described in Chapters 5, 6, or 7 to eliminate the actual redundancies.

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-nodes in blocks with incoming back edges, and therefore it must assign a unique value number to any ϕ-node in such a block.

```
procedure rename_and_value_number(Block b)
    Mark the beginning of a new scope
    for each φ-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 ← 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 φ-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 φ-node or assignment a in the original b
        for each name n defined by a
            pop S[n]
```

Figure 2.5 Value numbering during SSA construction

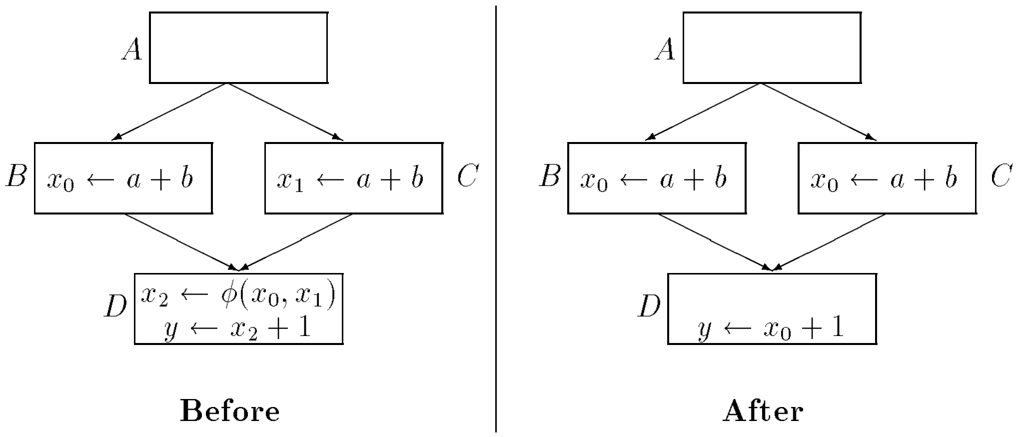


Figure 2.6 Unified hash table

2.5 Interaction with Other Optimizations

We should point out that eliminating more redundancies does not necessarily result in reduced execution time. This effect is a result of the way optimizations interact. The main interactions are with register allocation and combining instructions via copy 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 will discuss this issue further in Chapter 8.

The interaction between value numbering and other optimizations can also affect the execution time of the optimized program. The example in Figure 2.7 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

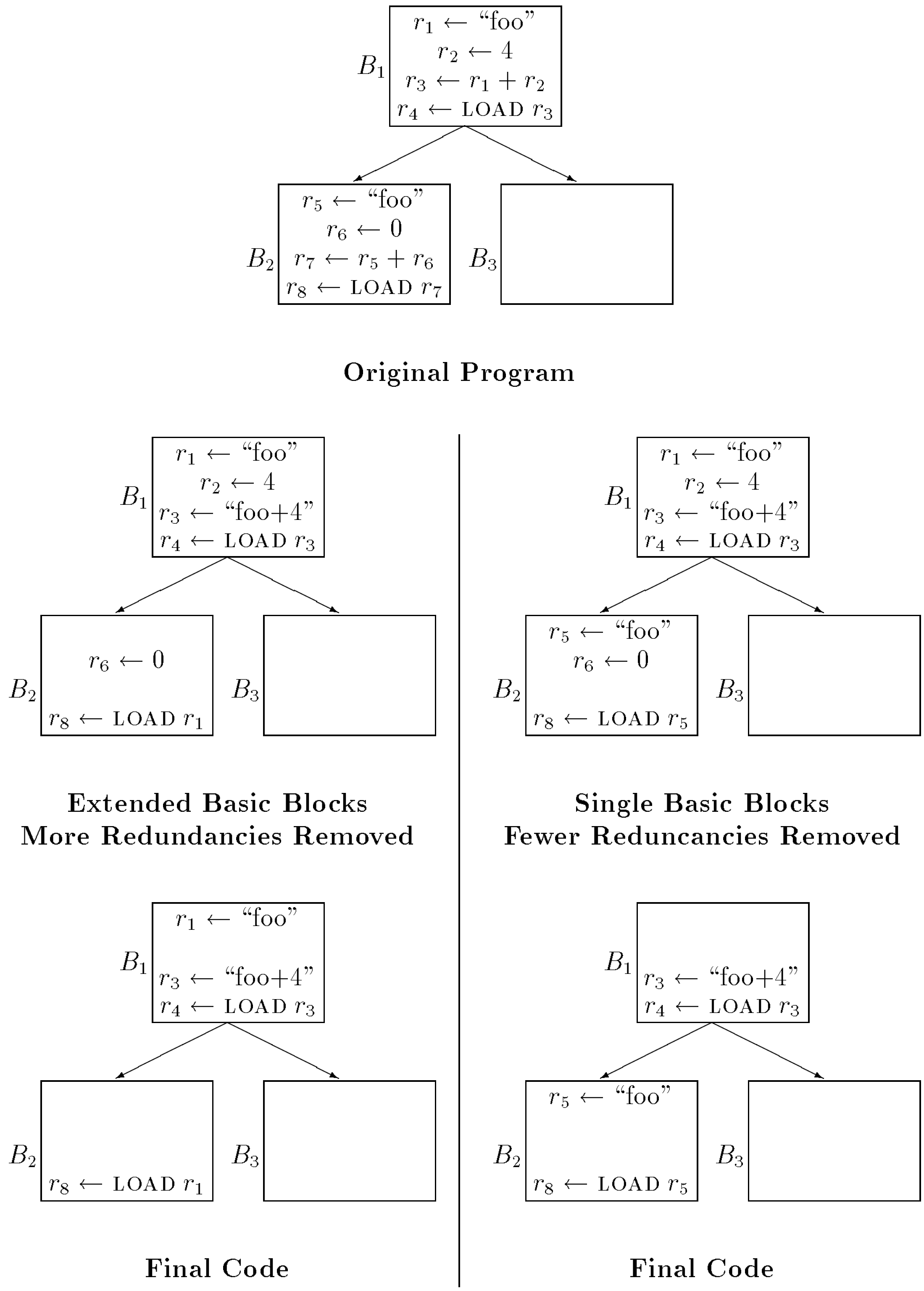


Figure 2.7 Interaction with other optimizations

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.

2.6 Data Structures

The data structures needed to support the hash-based value numbering algorithm in Figure 2.3 are relatively simple. The array VN maps SSA names to value numbers, and the hash table maps expressions to value numbers.

We maintain our hash tables using "overflow chaining". Further, since we value number during a walk of the dominator tree, we need a *scoped* hash table, similar to the style sometimes used to support compilation of a lexically-scoped language. To accomplish this, we a need an efficient way to remove the entries inserted during the processing of a particular block. Each entry in the table will contain pointers to two other entries:

1. the next entry in the same overflow chain, and
2. the next entry in the current scope.

At each nesting level, a variable called *scope* will point to the start of the list of entries in the current scope. Removing these entries is a simple matter of traversing this list. The hash table data structure is shown in Figure 2.8. In our implementation, there are two hash tables: one for expressions and one for ϕ-nodes.

2.7 Extensions

There are several extensions that can be applied to hash-based value numbering. A very simple extension is to handle commutative operators more flexibly. Whenever an expression with a commutative operator is encountered, we sort its operands based on their value numbers before searching the hash table. Thus, expressions containing a commutative operator are always put into a "canonical" form before processing. There are other operators that technically are not commutative, but they can also be handled more effectively using this technique. For example, $x < y$ should produce

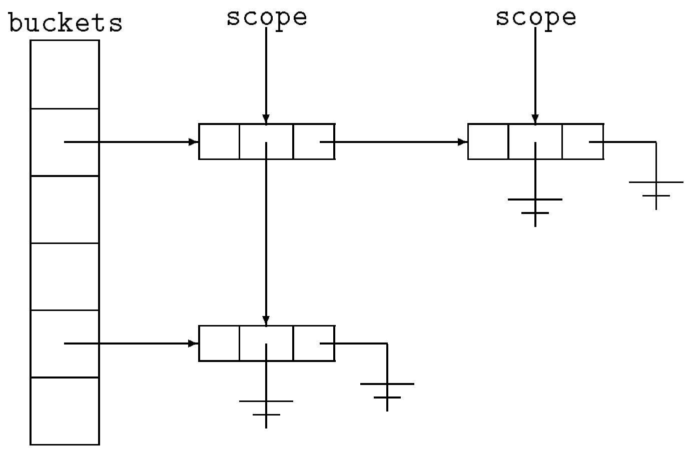


Figure 2.8 Data structure for scoped hash table

the same result as $y > x$; thus, we can place the expression in a canonical form by sorting the operands and reversing the sense of the operator if necessary.

We can significantly improve the effectiveness of the technique by incorporating constant folding and algebraic simplification. To accomplish this, the algorithm must keep track of which values represent compile-time constants and their values. Since the algorithm operates on SSA form, the only data structures required for this extension are

- a bit vector representing the values known to be constant, and
- an array mapping SSA names to their values (only those names with their entry set in the bit vector will have valid entries in this array).

Whenever an expression is found with constant operands, the expression is evaluated, and it is overwritten with the constant value. There are other simplifications that can be made when only one of the operands is a constant (*e.g.*, $x + 0 = x$), or when we know that two operands are equal (*e.g.*, $x - x = 0$). The complete list of simplifications used in our compiler is shown in Table 2.1.

Another improvement we can make to this technique is to trace values into and out of memory. This is accomplished via the *tags* in our intermediate representation. Each tag represents a distinct memory location, but certain tags may be aliased to each other. Operations that can affect memory are marked with a list of referenced

$$
\begin{aligned}
x - x &= 0 \\
x \text{ or } x &= x \text{ (bitwise)} \\
x \text{ xor } x &= 0 \text{ (bitwise)} \\
x \text{ and } x &= x \text{ (bitwise)} \\
x \text{ or } 0 &= x \text{ (bitwise)} \\
x \text{ xor } 0 &= x \text{ (bitwise)} \\
x \text{ and } 0 &= 0 \text{ (bitwise)} \\
\max(x, x) &= x \\
\min(x, x) &= x \\
\mathrm{SIGN}(x, x) &= x \text{ (Sign transfer}^2\text{)} \\
\mathrm{DIM}(x, x) &= 0 \text{ (Positive difference}^3\text{)} \\
x \stackrel{?}{=} x &= \mathrm{TRUE} \\
x \stackrel{?}{\leq} x &= \mathrm{TRUE} \\
x \stackrel{?}{\geq} x &= \mathrm{TRUE} \\
x \stackrel{?}{\neq} x &= \mathrm{FALSE} \\
x \stackrel{?}{<} x &= \mathrm{FALSE} \\
x \stackrel{?}{>} x &= \mathrm{FALSE} \\
x + 0 &= x \\
x - 0 &= x \\
x \times 0 &= 0 \\
\mathrm{SHIFT}(x, 0) &= x \\
0/x &= 0 \text{ (if } x \neq 0\text{)} \\
x \times 1 &= x \\
x/1 &= x \\
x \bmod 1 &= 0
\end{aligned}
$$

Table 2.1 Algebraic simplifications

[2] SIGN(x, y) = y $\geq$ 0 ? ABS(x) : -ABS(x)
[3] DIM(x, y) = x > y ? x - y : 0

tags, or a list of defined tags, or both. For LOAD and STORE operations, the first tag in each list is the one that appears in the source, and the others are possible aliases. During the conversion to SSA form, all tags are given subscripts to give each one a unique definition point. To trace values through memory, we must consistently hash LOAD and STORE operations. We will convert all LOAD opcodes into the corresponding STORE opcodes before hashing[4]. Memory operations are hashed using another table similar to the ones described in Section 2.6, called the "tag table".

We value number a LOAD operation by searching the hash table for a register that already contains the result. This can happen in two ways:

1. We could have previously loaded the value into a register.
2. We could have previously stored into this location from a register (without overwriting the location before we get to this LOAD).

If such a register is found, we use that register in place of the register being loaded. Otherwise, we add an entry into the tag table showing that this register contains the value at this memory location.

When we encounter a STORE operation, we want to determine if it is writing the same value to a location that is already there. This is often the case for the STOREs inserted just before subroutine calls. We look up the value stored in the memory location and compare it to the value being stored. If the two values are equal, the STORE operation can be deleted.

The example in Figure 2.9 illustrates how this process works. Notice that the LOADs into r_0 and r_1 will produce the same value. When the first LOAD is processed, we convert the LOAD operator into the corresponding STORE and search the hash table for the expression "STORE x_0". The expression will not be found, so an entry is added to the table with name r_0. When the next LOAD is processed, we will find the expression in the table and thus prove that $r_0 = r_1$. Next, the program stores a new value into location x. The STORE operation is marked with a list of both referenced and defined tags. The referenced list contains the "before" name of each tag, and the defined list contains the "after" name of each tag. If the value number of the "before" name is the same as the value number of the value being written to this memory location, then the STORE is redundant and can be removed. We add

[4]In our compiler, LOAD and STORE operations are distinguished by the type of the value and the addressing mode

$$
\begin{array}{ll}
r_0 \leftarrow \text{LOAD} & [x_0] \\
r_1 \leftarrow \text{LOAD} & [x_0] \\
\quad \vdots & \\
r_2 \leftarrow \ldots & \\
\text{STORE } r_2 & [x_0][x_1] \\
r_3 \leftarrow \text{LOAD} & [x_1] \\
\quad \vdots & \\
\text{STORE } r_3 & [x_1][x_2]
\end{array}
$$

Figure 2.9 Example program containing LOADs and STOREs

the expression "STORE x_1" with the name r_2 to the table. This entry is then used to prove that $r_2 = r_3$. Finally, the STORE into x from r_3 can be deleted because the expression "STORE x_1" and r_3 have the same value number, namely r_2.

For simplicity, we have ignored the address calculation used to access the variable in our example. For scalar variables it is irrelevant. On the other hand, the address must be considered when processing array variables. In reality, we consider the address to be another operand of the STORE expression (*e.g.*, STORE *addr* a_0). By doing this, we can distinguish between accesses to different elements of an array. Assume that the variable x in Figure 2.9 is an array rather than a scalar. If each access is to the same element, value numbering may be able to determine that the addresses for each access are equal, and the redundancies will be eliminated as before. However, if each access is to a different element, the address operand of the expressions will not match, and the program will remain unchanged.

2.8 Summary

This chapter describes the hash-based approach to value numbering. The technique was first applied to single basic blocks and later to extended basic blocks. We have shown that the use of static single assignment form simplifies the technique as well as provides opportunities for further improvements. We have shown new algorithms for hash-based value numbering over a routine's dominator tree and using a unified hash table. When applied to single basic blocks, extended basic blocks, or the dominator tree, value numbering is performed on-line – redundancies are removed as soon as they are identified. The unified hash table approach is an off-line algorithm – it relies on the techniques described in Chapters 5, 6, or 7 to eliminate the redundancies. Another contribution of this chapter is the ability to trace values through memory and to remove any redundancies identified.

Hash-based value numbering has several advantages. It is easy to understand and to implement; it can easily handle constant folding and algebraic identities, and it runs in expected linear time. The unified hash table algorithm is almost global, but it can fail to discover redundancies where values flow through a back edge in the CFG.

Chapter 3

Value Partitioning

Alpern, Wegman, and Zadeck describe a global approach to value numbering that we call *value partitioning* [4]. In this chapter, we extend this algorithm to handle commutative operations and to eliminate redundant STORE operations. The algorithm operates on the static single assignment (SSA) form of the routine [18]. In contrast to hash-based approaches, this technique partitions values into congruence classes. Two values are *congruent* if they are

1. Computed by the same opcode, and

2. Each of the corresponding operands are congruent.

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. The maximal fixed point partitions the values in the routine into congruence classes. The primary advantage of value partitioning (over hash-based value numbering) is that it is a global algorithm. However, it only accomplishes the first objective of value numbering (See Chapter 2). It does not accomplish constant folding or algebraic simplification, and it does not remove redundant computations. Redundant computations are removed by a separate algorithm.

We partition the values in the routine into congruence classes using a variation of the algorithm by Hopcroft for minimizing a finite-state machine [1]. We use the term *partition* to refer to a set of *congruence classes* such that each value (SSA name) in the routine is in exactly one class. The partitioning of values is accomplished by starting with an initial partition and iteratively refining the partition until it stabilizes. In the initial partition, all values defined by the same opcode are in the same congruence class. Of course, the set of values defined by some opcodes must be divided immediately. For example, the FRAME opcode in our compiler defines a set

```
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 3.1 Partitioning algorithm

of distinct values that are passed to the routine in registers. Therefore, all the values defined by the FRAME opcode are placed in different classes in the initial partition.

The partition is refined using a worklist algorithm. The variable called *worklist* will contain classes that might cause other classes to split. When a class is removed from *worklist*, we visit the uses of all its members. The *touched* set records the values defined by the uses that are visited at each iteration. Notice that we only visit uses in the same position because we must treat all opcodes as if they were not commutative. Commutative operations are handled by an extension to the partitioning algorithm described in Section 3.4. Any class s (s stands for split) with a proper subset of its members in *touched* ($\emptyset \subset (s \cap touched) \subset s$) must be split into two classes. We create a new class n containing the touched members of s, and we remove the members of n from s. Splitting class s may cause other classes in the partition to split, so we must update *worklist*. There are two cases to consider:

Case 1 If s was already in *worklist*, this means that the partition was not stable with respect to s, so we must leave s in *worklist* and also add n to *worklist*.

Case 2 If s was not in *worklist*, this means that the partition was stable with respect to s before it was split. Thus, any class that would be split as a result of

> removing s from *worklist* would be split the same way as a result of removing n from *worklist*. Since we are free to choose between s and n, we will select the one that can be processed in a smaller amount of time. Therefore, we add the smaller of s and n to *worklist*. As we shall see, this step is fundamentally important in achieving our desired running time.

Once we have refined the partition with respect to class c, we will not examine c again unless it is split. Pseudo-code for the partitioning algorithm is shown in Figure 3.1.

To illustrate how the algorithm operates, consider the code fragment in Figure 3.2. Figure 3.3 shows the initial partition if X is not congruent to Y ($X \ncong Y$). Notice that all names defined by the subtraction opcode are initially in the same class. Let the class containing X be the first removed from *worklist*. Touching the uses of X in position 1 will result in A being removed from its class. Touching the uses of X in position 2 will result in B being removed from its class. The partition after one iteration is shown in Figure 3.3. Let the class containing A be the next one removed from *worklist*. Touching the uses of A in position 1 will result in C being removed from its class. Touching the uses of A in position 2 will not cause any classes to be split because the *touched* set is the entire class. Since all values are now in different classes, no further splitting can occur. However, since the algorithm has no way to determine this, it will not terminate until *worklist* is empty. The final partition is also shown in Figure 3.3.

To understand the importance of touching only uses in the same position, consider again the code fragment in Figure 3.2 and its initial partition in Figure 3.3. Touching all uses of X in any position will result in A and B being removed from their class and put into a new class together. If the algorithm continues in this fashion, no further splitting will occur. The final partition is shown in Figure 3.4. Notice that we have erroneously proven that $A \cong B$ and $C \cong D$.

$$\begin{aligned} A &\leftarrow X - Y \\ B &\leftarrow Y - X \\ C &\leftarrow A - B \\ D &\leftarrow B - A \end{aligned}$$

Figure 3.2 Partitioning example

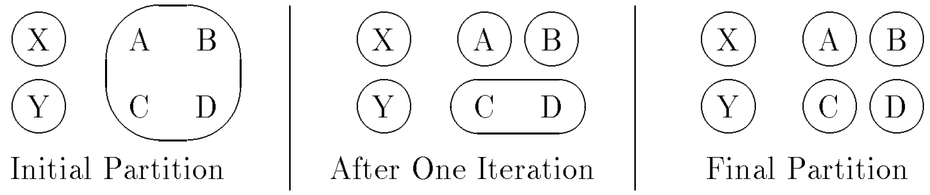


Figure 3.3 Partitioning steps for example program

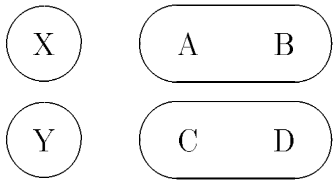


Figure 3.4 Incorrect partition when positions are ignored

3.1 Complexity

From the pseudo-code given in Figure 3.1, it is not clear what the running time of the algorithm will be. We claim that the partition can be computed in $\mathbf{O}(E \log N)$ time, where E is the number of edges and N is the number of nodes in the routine's SSA graph[5]. However, only a careful implementation of the data structures will result in the desired time bound. To discover the operations our data structure must support and their complexities, we will analyze the algorithm from the bottom up. The statement in line 15 can be performed in constant time if we can determine the size of a class in constant time. Therefore, the **if** statement in lines 12–15 will require constant time. The splitting of a class in lines 10 and 11 can be performed in $\mathrm{O}(\|n\|)$ time if we can remove an element from class s and insert it into class n in constant time[6]. The crucial part of the implementation is to perform the **for** loop in lines 9–15 in $\mathrm{O}(\|touched\|)$ time. This will be discussed in detail in Section 3.3. The **for** loop in lines 7 and 8 can be performed in $\mathrm{O}(\|c\|)$. We assume that the number of uses in any operation is bounded by a constant so the maximum position of a use is also bounded by a constant. Therefore, we can ignore the **for** loop starting on line 5.

[5]The SSA graph has a node for each definition and edges flow from definitions to uses.

[6]We use the notation $\|n\|$ to represent the size of set n.

Operation	Complexity
Determine the number of elements in a class	O(1)
Determine which class contains a name	O(1)
Remove a member of a class	O(1)
Add a member to a class	O(1)
Add a new class to the partition	O(1)
Iterate through the members of class c	O($\|c\|$)

Figure 3.5 Operations supported by the partition

Now we are ready to analyze the running time of the entire algorithm. Consider the number of times a class containing a particular element x can be removed from *worklist*. Each time such a class is chosen, it must be smaller than half the size of the last class to contain x removed from *worklist*. This is because we only add the smaller of n and s in line 15. Therefore, a class containing x can be removed from *worklist* at most $\mathbf{O}(\log N)$ times. Suppose the cost of each execution of the **for** loop in lines 9–15 is charged to each x in c according to the number of uses of x in position p. Then the cost for each x is $\mathbf{O}(\|\mathrm{USES}(x)\| \log N)$, where USES($x$) is the set of uses of x. Since E is the set of all uses of any x, the total cost of the algorithm is $\mathbf{O}(E \log N)$. Figure 3.5 summarizes the time complexity required for the various operations that will be performed on the partition.

3.2 Data Structures to Support Partitioning

To support the complexity requirements given in Figure 3.5, each class will contain the following information:

1. the number of members, and

2. a doubly-linked list of members.

We will also maintain a lookup table that will have a pointer to the list node and the class number of each element in the partition. This will enable us to locate and remove an element from a class in constant time. Figure 3.6 shows how the data structure would look assuming that the item with SSA name x_0 is in class 5.

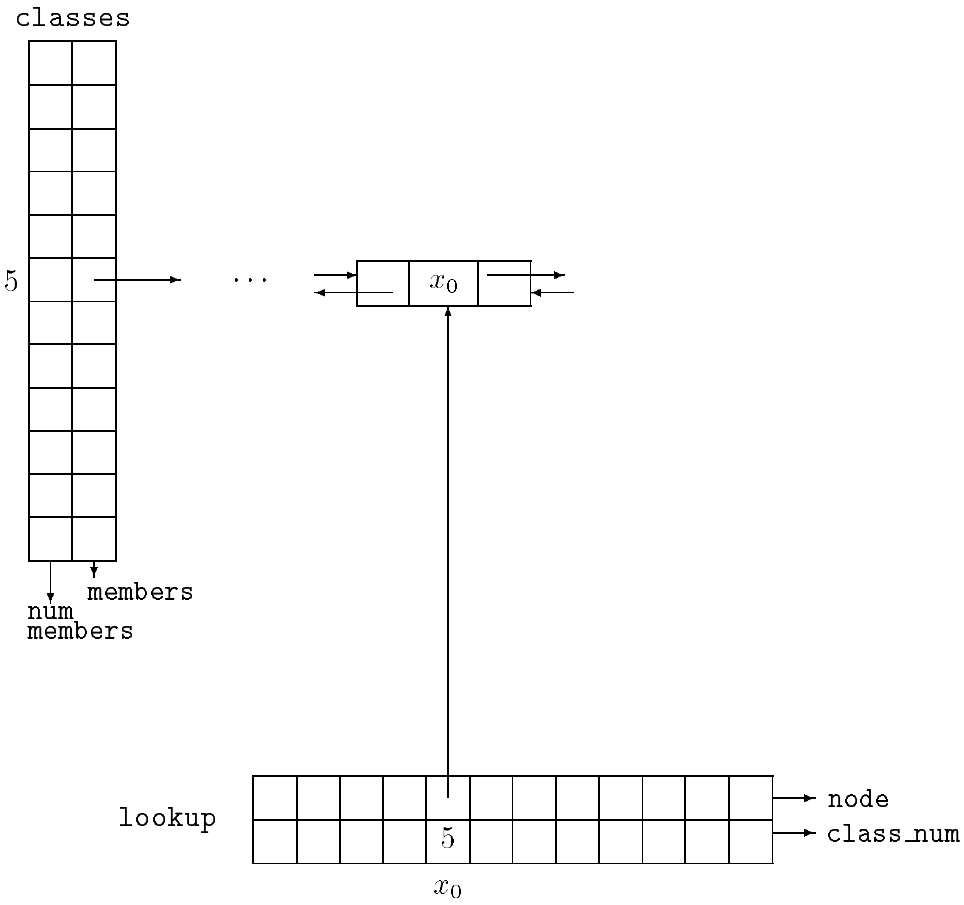


Figure 3.6 Data structures for representing the partition

3.3 Refining the Partition

Recall that in our initial presentation of the partitioning algorithm (Figure 3.1) we used a set called *touched*. This set contained all uses of members of a class in some position. We then searched for classes with a proper subset of their members in *touched* and split them[7]. An implementation of the algorithm must employ an efficient technique for determining which classes to split and which members to remove at each iteration. In Section 3.2, we claimed that this step could be performed in O($\|touched\|$) time. To accomplish this, we do not represent the *touched* set itself. Instead, we keep the intersection of *touched* and each class in the partition. This information is kept in an array called `intersections`. Each element of `intersections` will contain the intersection of *touched* with the corresponding element of `classes`. We must also keep track of which classes have a non-empty `intersections` entry. When an item is touched during the process of refining the partition, it is moved out of its class and into the `intersections` array. Recall that we are only interested in classes with a proper subset of their members touched. Therefore, if all members of a class are touched, they will be returned to their original class.

Figure 3.7 depicts an example partition with SSA name x_0 is in class number 5. The entry in `lookup`[x_0] indicates class number 5 and points to the node for x_0 in the `members` list of `classes[5]`. If x_0 is touched, the node will be removed from that list and appended to the `members` list of `intersections[5]`. Once all the uses in position p have been touched, we can determine which classes must be split by iterating through each class s with a non-empty list at `intersections`[s]. To accomplish this, we must carefully maintain the set of classes to be split. We have already removed the necessary members from `classes`[s] and placed them in `intersections`[s]. Thus, we can simply create a new class containing the members of `intersections`[s] and iterate through the members list, updating the `class_num` field of the `lookup` array. The entire process requires O($\|touched\|$) time.

3.4 Handling Commutative Operations

Commutative operations must be handled with an extension to the partitioning algorithm. The idea is to ignore the position when touching a use of a member of a congruence class. Now, instead of splitting classes based on which members were

[7] In other words, we split all classes s such that $\emptyset \subset (s \cap touched) \subset s$.

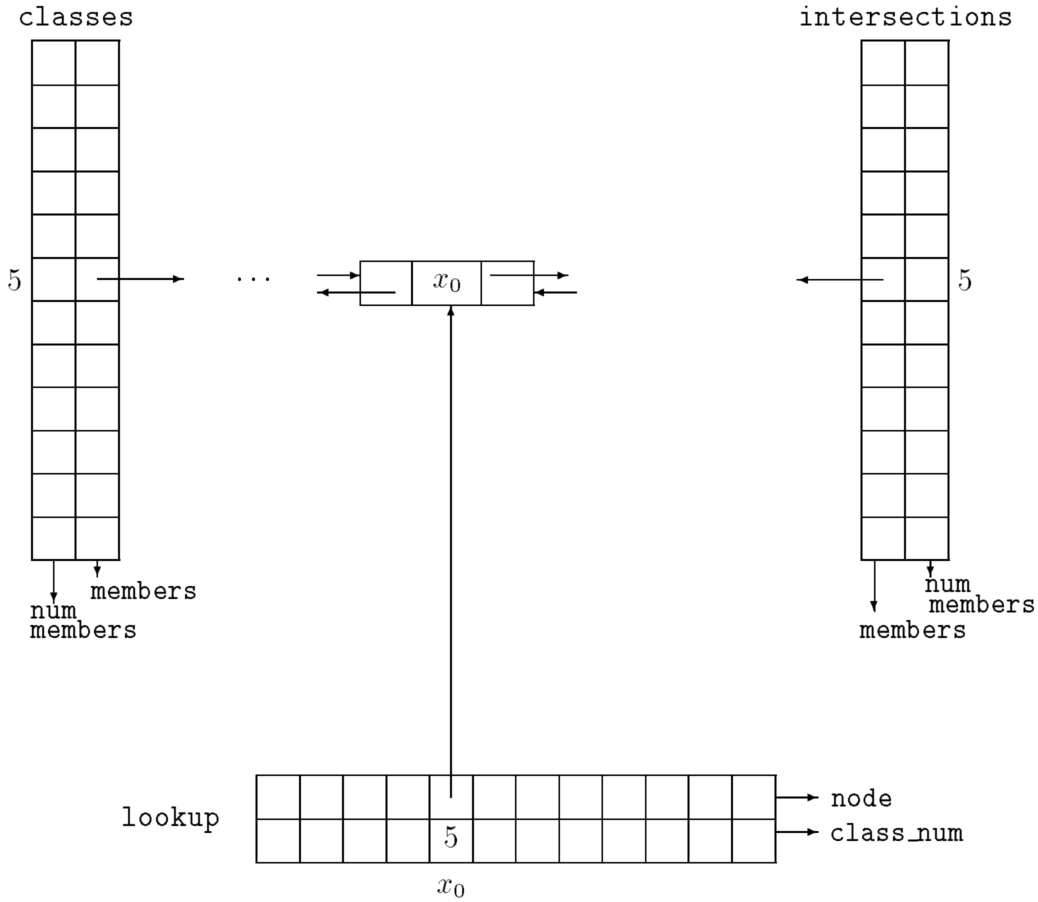


Figure 3.7 Data structures for refining the partition

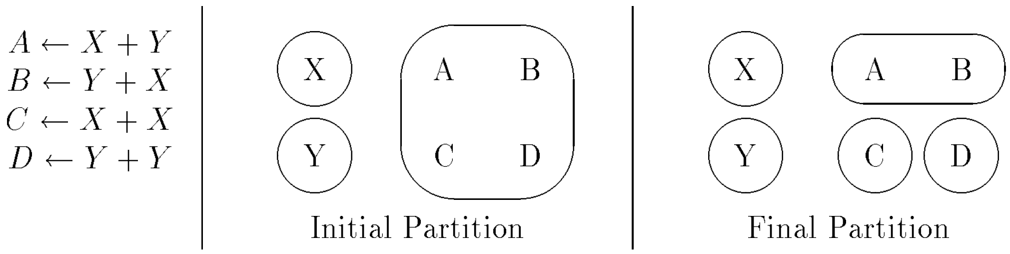


Figure 3.8 Commutativity example

touched or not touched, we split classes based on which members were touched 0, 1, or 2 times. Consider the example code fragment in Figure 3.8, and assume that $X \ncong Y$. The initial partition is shown in Figure 3.8. Let the class containing X be the first removed from the worklist. Touching the uses of X will result in A and B being touched once and C being touched twice. Therefore, A and B will be placed in a class together, and C will be in a class by itself. Let the class containing Y be the next one removed from the worklist. Touching the uses of Y will result in A and B being touched once and D being touched twice. Therefore, no further splitting of classes is required. The final partition is shown in Figure 3.8.

Now assume that $X \cong Y$. The initial partition is shown in Figure 3.9. Let the class containing X and Y be the first removed from the worklist. Touching the uses of X and Y will result in A, B, C, and D all being touched twice. Therefore, no further splitting of classes is required, and the initial partition is also the final partition.

Recall that when we touched an item for non-commutative operations, we moved the item out of its current class and into the `intersections` array. The `touched_once` and `touched_twice` arrays are used like the `intersections` array. The first time an item is touched, it is moved from its original class into `touched_once`. The second time it is touched, it is moved from `touched_once` to `touched_twice`. We must be careful to keep track of which classes have a non-empty entry in `touched_once` or `touched_twice`, just as we do for the `intersections` array.

Figure 3.10 depicts the partition if the variable A in Figure 3.8 is in class number 5. The first time A is touched, it will be removed from the `members` list of `classes[5]` and appended to the `members` list of `touched_once[5]`. The second time A is touched, it will be removed from the `touched_once[5]` and appended to `touched_twice[5]`. Each time the item is touched, the `num_touches` field of the `lookup` array will be incremented.

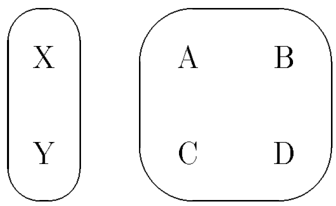


Figure 3.9 Partition for second commutativity example

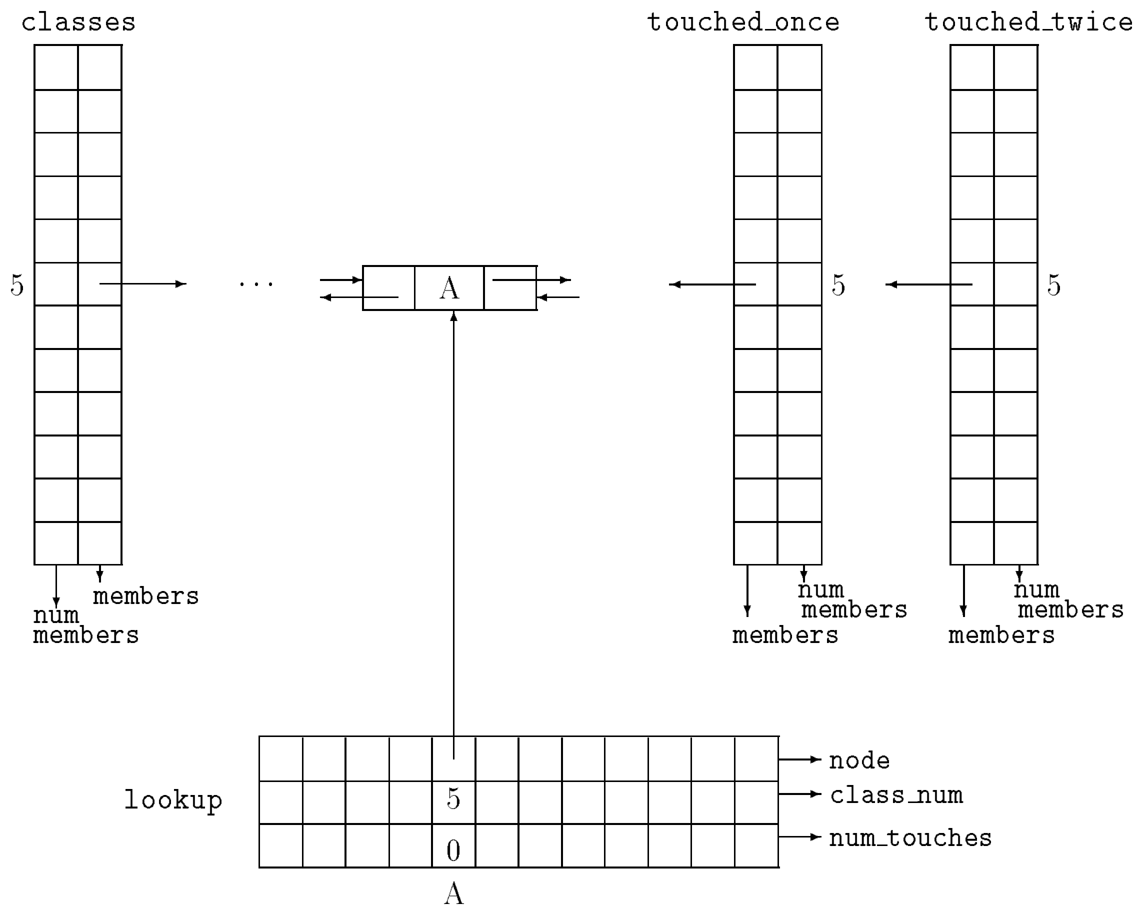


Figure 3.10 Data structures for handling commutative operations

3.5 Eliminating Redundant Stores

Redundant-store operations write the same value to a memory location that was previously written there. Therefore, they do not alter the contents of memory, and they can be eliminated from the routine, but this requires an extension to the original algorithm. Redundant stores should not be confused with dead stores, which write a value to memory that is never subsequently read. Handling redundant scalar stores requires an extension to the partitioning algorithm. Consider the program fragment in Figure 3.11. The FRAME operation defines the initial values for the memory tags x and y. During the conversion to SSA form, all tags were given subscripts to give each one a unique definition point. The STORE operations have two tags associated with them[8]. The first tag is the "before" value of the memory location, and the second is the "after" value of the memory location. In our example, the STORE from r_0 is redundant while the STORE from r_1 is not. We would like to check for congruence between the before and after tag values to determine if a STORE is redundant. Unfortunately, the tag x_0 is defined by the FRAME operation, and the tag x_1 is defined by a STORE operation. Therefore, they will be in different classes in the initial partition. The initial partition is shown in Figure 3.12. Clearly, x_0 can never be congruent to x_1 using the unmodified partitioning algorithm.

We must treat scalar LOAD and STORE operations as copies from a register to memory or vice versa. Since copying a value does not change it, we must ensure that the source and destination of a copy will remain in the same congruence class throughout the partitioning process. To accomplish this, we keep a list of copies for

FRAME $[x_0\ y_0]$
$\vdots$
$r_0 \leftarrow$ LOAD x_0
r1 $\leftarrow$ 1
$\vdots$
STORE r_0 $[x_0]\ [x_1]$
STORE r_1 $[y_0]\ [y_1]$

Figure 3.11 Example program for redundant-store elimination

[8]Actually, these are lists of tags. The first tag is the one that appears in the source, and the others are possible aliases. In this example, the lists have length one.

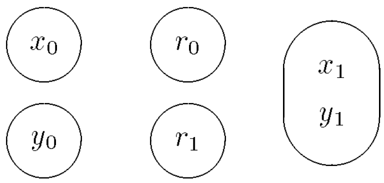


Figure 3.12 Initial partition for redundant-store elimination example

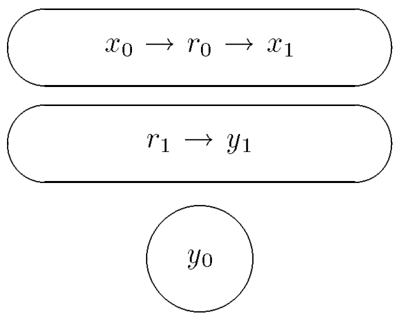


Figure 3.13 Partition to enable redundant-store elimination

each item in the partition. During the process of refining the partition, the copy list for an item moves from class to class with the original item. In our example routine, r_0 is treated as a copy of x_0, and x_1 is treated as a copy of r_0. Also, y_0 is a copy of r_1. Given this scheme, the initial partition is also the final partition (See Figure 3.13). Notice that $x_0 \cong x_1$, but $y_0 \ncong y_1$. Thus, we can eliminate the first STORE but not the second.

Figure 3.14 depicts the data structures representing the partition if x_0 from Figure 3.11 is class number five. The items r_0 and x_1 are in the copy list for x_0. Notice that the entries in `lookup` for r_0 and x_1 point to the node for x_0, and the $\mathcal{T}$ entry in the `is_copy` field indicate that they are copies. This tells the partitioning algorithm to ignore them during the refinement process.

3.6 Summary

This chapter describes in detail the value partitioning algorithm of Alpern, Wegman, and Zadeck. This algorithm uses a variation of Hopcroft's DFA minimization algorithm to partition the values in a routine into congruence classes. The implementation of value partitioning is described in detail. We present extensions to the algorithm to handle commutative operations and to eliminate redundant store operations.

When compared to hash-based value numbering, value partitioning has the advantage that it is global. However, value partitioning has some disadvantages. It is difficult to implement; it cannot handle constant folding and algebraic identities, and it runs in $\mathbf{O}(E \log N)$ time.

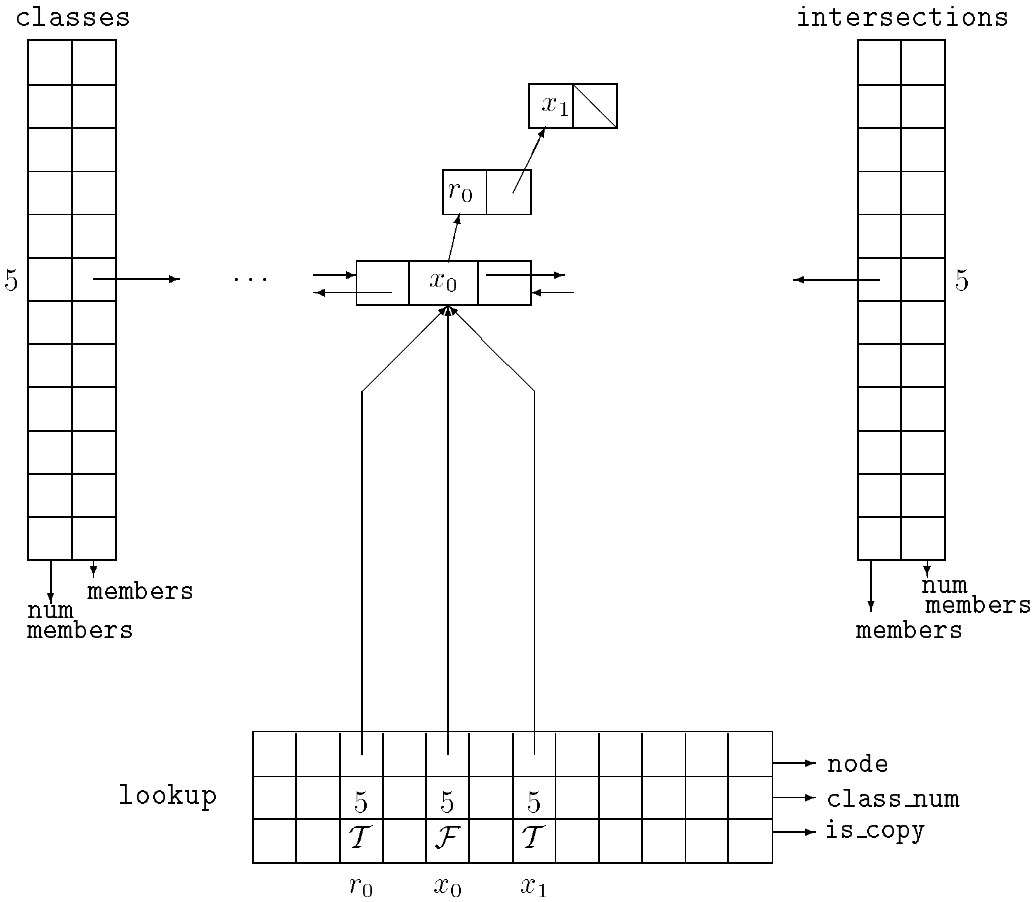


Figure 3.14 Data structures for redundant-store elimination

Chapter 4

SCC-Based Value Numbering

In Chapters 2 and 3, we described two competing value numbering techniques: 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 chapter, 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.

4.1 Shortcomings of Previous Techniques

Assume that X and Y are known to be equal in the code fragment in Figure 4.1. 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.

$$
\begin{aligned}
A &\leftarrow X - Y \\
B &\leftarrow Y - X \\
C &\leftarrow A - B \\
D &\leftarrow B - A
\end{aligned}
$$

Figure 4.1 Improved by hash-based techniques

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 Figure 4.2. 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.

The shortcomings of hash-based value numbering and value partitioning suggest a need for a value numbering algorithm that combines the advantages of both 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. Click presents an extension to value partitioning that includes constant folding, algebraic simplification, and unreachable code elimination [13]. 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

$$
\begin{array}{l}
X_0 \leftarrow 1 \\
Y_0 \leftarrow 1 \\
\text{while } (\ldots) \\
\qquad X_1 \leftarrow \phi(X_0, X_2) \\
\qquad Y_1 \leftarrow \phi(Y_0, Y_2) \\
\qquad X_2 \leftarrow X_1 + 1 \\
\qquad Y_2 \leftarrow Y_1 + 1
\end{array}
$$

Figure 4.2 Improved by partitioning techniques 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.

This chapter presents an algorithm, called SCC-based value numbering, that more closely resembles hash-based value numbering than value partitioning. 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 [32]. 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 understand. 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.

4.2 The RPO Algorithm

The algorithm in Figure 4.3 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 $x[i]$ represents 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.*, $\langle x_0 \rangle$). The *lookup* function

for all SSA names i
    VN[i] $\leftarrow \top$
do
    changed $\leftarrow$ `FALSE`
    for all blocks b in reverse postorder
        for all definitions x in b
            expr $\leftarrow \langle \mathrm{VN}[x[1]], x.op, \mathrm{VN}[x[2]] \rangle$
            temp $\leftarrow$ lookup(*expr*, x)
            if VN[x] $\neq$ *temp*
                changed $\leftarrow$ `TRUE`
                VN[x] $\leftarrow$ *temp*
    Remove all entries from the hash table
while *changed*

Figure 4.3 The RPO algorithm

searches a hash table for the expression $\langle \mathrm{VN}[x[1]], x.\mathrm{op}, \mathrm{VN}[x[2]] \rangle$. 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 $\mathrm{VN}[x] = \mathrm{VN}[y]$.

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

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

We refer to a partition by the equivalence relation that produces it. We say that $\cong_i$ is a refinement of $\cong_j$ ($\cong_i \preceq \cong_j$) 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 4.1 $x \cong_i y \Rightarrow x \cong_{i-1} y$

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 – $x \not\cong_{i-1} y$ and $x \cong_i y$. Consider the reasons why $x \not\cong_{i-1} y$:

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 4.1 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:

$$x \cong y \quad \text{iff} \left\{ \begin{array}{l} x.\mathrm{op} = y.\mathrm{op} \\ x[e] \cong y[e], \quad \forall e \end{array} \right.$$

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 post-order, 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 4.1 If $x \ncong_i y$ 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:

$$\begin{array}{ccc} x & \ncong_i & y \\ x[e_1] & \ncong_{i-b_1} & y[e_1] \\ & \vdots & \\ x[e_1]\ldots[e_n] & \ncong_{i-b_n} & y[e_1]\ldots[e_n] \end{array}$$

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 $x \ncong_i y$ 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 $x \ncong_i y$:

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

Case 2 $(x[e] \ncong_i y[e])$ **for some non-back edge)** This 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 $(x[e] \ncong_{i-1} y[e])$ **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 $x \ncong_i y$:

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

Case 2 ($x[e] \ncong_i y[e]$) 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 ($x[e] \ncong_{i-1} y[e]$) 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 4.2 $x \cong_{D(\text{SSA})+1} y \Rightarrow x \cong_{D(\text{SSA})+2} y$

Proof. Suppose not – let x be the SSA name with the smallest RPO number such that $x \cong_{D(\text{SSA})+1} y$ and $x \ncong_{D(\text{SSA})+2} y$. According to Lemma 4.1, there is a sequence of inputs such that:

$$\begin{array}{rcl} x & \ncong_{D(\text{SSA})+2} & y \\ x[e_1] & \ncong_{D(\text{SSA})+2-b_1} & y[e_1] \\ & \vdots & \\ x[e_1]\ldots[e_n] & \ncong_1 & y[e_1]\ldots[e_n] \end{array}$$

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, $x \ncong_i y$ for some $i < D(\text{SSA}) + 2$. By Theorem 4.1, $x \ncong_{D(\text{SSA})+1} y$, a contradiction. □

Corollary 4.2 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. □

4.3 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 [45]. 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 $D(\mathrm{SSA})+1$ 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 $2D(\mathrm{SSA})+2$. However, since each element falls in both frameworks on the first iteration, any element can fall at most $2D(\mathrm{SSA})+1$ times. Therefore, the extended algorithm must terminate in $2D(\mathrm{SSA})+2$ iterations.[9]

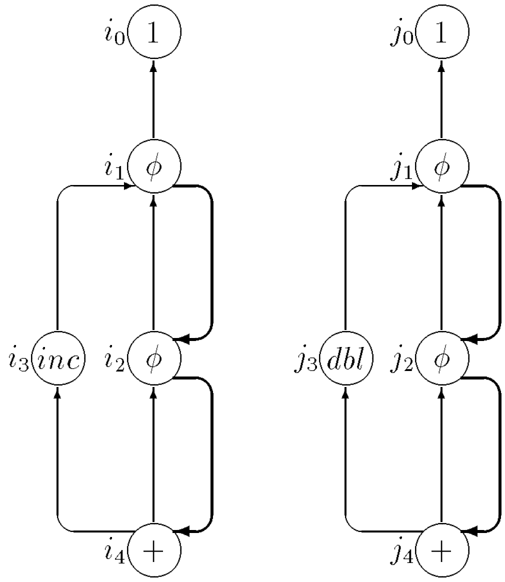
The example in Figure 4.4 requires $2D(\mathrm{SSA})+2$ iterations for the algorithm to terminate. The back edges are shown with bold arrows. After the first iteration, all nodes are believed to be constants; notice that both i_3 and j_3 are assigned the constant 2. During the next iteration, i_2, j_2, 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 $i_4 \ncong j_4$. On the fourth and fifth iteration, we prove that $i_2 \ncong j_2$ and $i_1 \ncong j_1$, respectively. On the sixth iteration, the partition stabilizes and the algorithm terminates. Intuitively, the algorithm takes $D(\mathrm{SSA})$ passes to prove that i_3 and j_3 are not the constant 2, and thus cannot be equal; then it takes another $D(\mathrm{SSA})$ passes to propagate this fact.

4.4 Discussion

We have shown that the RPO algorithm finds at least as many congruences as hash-based value numbering or value partitioning in $\mathbf{O}(N \times D(\mathrm{SSA}))$ time. Kam and Ullman showed that the iterative data-flow analysis used for a large class of data-flow frameworks requires $D(\mathrm{CFG})$ passes over the CFG [27]. 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 [32]. We expect that for most programs $D(\mathrm{CFG}) = D(\mathrm{SSA})$. However, the program in Figure 4.5 is an example where this is not true. The back edges are shown with bold arrows. Notice that $D(\mathrm{CFG}) = 2$, but $D(\mathrm{SSA}) = 6$. Further, we could make $D(\mathrm{SSA})$ 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.

4.5 The SCC Algorithm

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



SSA Graph

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

Value Numbers

Figure 4.4 Example requiring $2D(\mathrm{SSA}) + 2$ iterations

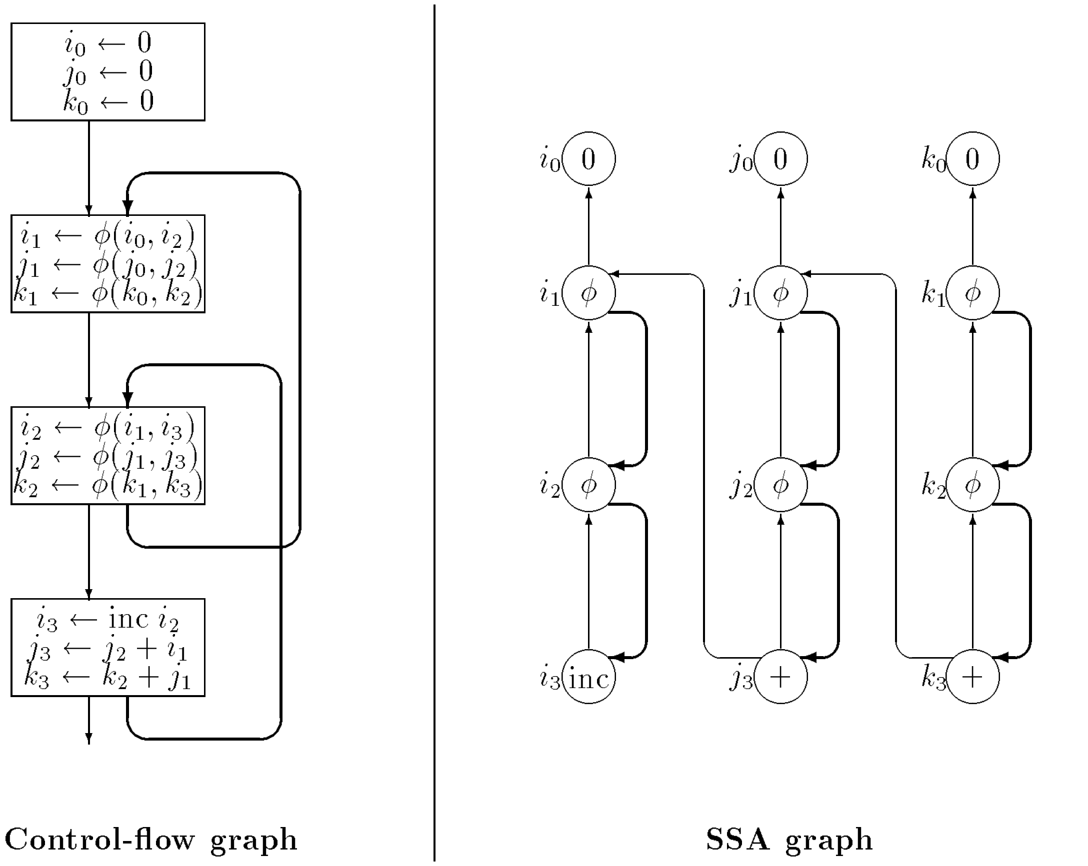


Figure 4.5 Example with $D(\text{CFG}) \neq D(\text{SSA})$

To make the algorithm more efficient in practice, we operate on the SSA graph instead of the control-flow graph. If no cycles are present in the SSA graph, we can simply process the nodes in a single reverse postorder walk. That ordering guarantees that all operands of an expression are visited before the expression itself must be processed. If cycles are present then no such ordering exists. We can identify the *strongly connected components* (SCCs) of the graph and treat each SCC as a single node. A strongly connected component is a maximal collection of nodes such that given any pair of nodes in the collection there is a path between them. After collapsing each SCC into a single node, the resulting graph must be acyclic. When we traverse the nodes of this graph in reverse postorder, we know that all operands of a node are processed before the node itself. A node that is not part of any cycle requires no special processing, but a node that represents a strongly connected com-

```
DFS(node)
    node.DFSnum ← nextDFSnum++
    node.visited ← TRUE
    node.low ← node.DFSnum
    PUSH(node)
    for each o ∈ {operands of node}
        if not o.visited
            DFS(o)
            node.low ← MIN(node.low, o.low)
        if o.DFSnum < node.DFSnum and o ∈ stack
            node.low ← MIN(o.DFSnum, node.low)
    if node.low = node.DFSnum
        SCC ← ∅
        do
            x ← POP()
            SCC ← SCC ∪ {x}
        while x ≠ node
        ProcessSCC(SCC)
```

Figure 4.6 Tarjan's SCC finding algorithm

ponent requires special processing. This observation led us to an improved algorithm, called SCC-based value numbering 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, shown in Figure 4.6 [42]. Tarjan's 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 process the nodes as they 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.

```
initialize optimistic and valid tables
for all nodes n
    n.valnum ← ⊤

while there is an unvisited node n
    DFS(n) (see Figure 4.6)


ProcessSCC(SCC)
    if SCC has a single member n
        Valnum(n, valid)
    else
        do
            changed ← FALSE
            for each n ∈ SCC in reverse postorder[10]
                Valnum(n, optimistic)
        while changed
        for each n ∈ SCC in reverse postorder
            Valnum(n, valid)


Valnum(node, table)
    expr ← ⟨node[1].valnum, node.op, node[2].valnum⟩
    Try to simplify expr
    temp ← lookup(expr, table, node.SSAname)
    if node.valnum ≠ temp
        changed ← TRUE
        node.valnum ← temp
```

Figure 4.7 SCC-based value numbering algorithm

[10]Reverse postorder numbers are assigned with respect to the control-flow graph

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 *valid* table contains only facts that are known to be true, while the *optimistic* table contains facts that may later be disproven. Single nodes are processed once using the *valid* table, and collections of nodes are processed iteratively using the *optimistic* table. Figure 4.7 shows the algorithm for SCC-based value numbering. The first step is to initialize the *optimistic* and *valid* tables and to assign $\top$ as the value number for each node in the SSA graph. A value number of $\top$ is considered congruent to everything; it indicates that this node has not yet been examined. Next, we repeatedly apply Tarjan's algorithm to any unvisited node in the graph. As Tarjan's algorithm identifies strongly connected components, it calls ProcessSCC. This function decides if the SCC is a single node or a collection of nodes. Single nodes are processed using the valid table. Collections of nodes are processed by iterating in reverse postorder (with respect to the CFG) using the *optimistic* table. After the iteration stabilizes, the nodes are processed one final time using the *valid* table. The Valnum function processes each node. It first tries to simplify the expression that the node represents. For ordinary instructions, we perform constant folding and algebraic simplification. For ϕ-nodes, simplification can be performed if all the operands are equal. We can also simplify ϕ-nodes of the form $\phi(\langle x\rangle, \top)$ to $\langle x\rangle$[11]. Intuitively, this allows the initial value of a variable to "fall through" the ϕ-node and enter the loop. Since we are iterating in reverse postorder, $\top$ can only appear as an argument to a ϕ-node – not as an operand of an instruction.

4.6 Example

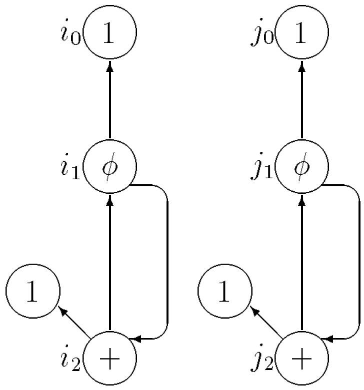
To further clarify the algorithm, consider how it would proceed if given the example in Figure 4.8. The values i_0 and j_0 are not contained in any cycle, so they will be assigned value numbers before either of the SCCs. Assume they are each given the value number $\langle i_0\rangle$ and that the SCC containing i_1 and i_2 is processed next. During the first pass over the SCC, the ϕ-node $\phi(\langle i_0\rangle, \top)$ will be simplified (optimistically) to $\langle i_0\rangle$, and i_1 will be given value number $\langle i_0\rangle$.[12] Then, the expression defining i_2, $\langle i_0\rangle + 1$, can be simplified to 2. An entry mapping the constant 2 to value number $\langle i_2\rangle$ will be added to the *optimistic* table. During the second pass, the expression

[11] Recall that $\top$ is considered congruent to everything

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

$i_0 \leftarrow 1$
$j_0 \leftarrow 1$
while $(\dots)$
  $i_1 \leftarrow \phi(i_0, i_2)$
  $j_1 \leftarrow \phi(j_0, j_2)$
  $i_2 \leftarrow i_1 + 1$
  $j_2 \leftarrow j_1 + 1$

SSA Form



SSA Graph

Figure 4.8 Example with equal induction variables

$\phi(\langle i_0 \rangle, \langle i_2 \rangle)$ cannot be simplified, so an entry mapping the expression to $\langle i_1 \rangle$ is added to the *optimistic* table. Next, an entry mapping $\langle i_1 \rangle + 1$ to $\langle i_2 \rangle$ will be added to the *optimistic* table. At this point the value numbers have stabilized, so we make a final pass using the *valid* table. During this pass, we add entries mapping $\phi(\langle i_0 \rangle, \langle i_2 \rangle)$ to $\langle i_1 \rangle$ and mapping $\langle i_1 \rangle + 1$ to $\langle i_2 \rangle$ to the *valid* table. Notice that the optimistic entry mapping 2 to $\langle i_2 \rangle$ is not added to the *valid* table - this assumption has been disproven.

The next step is to process the SCC containing j_1 and j_2. During the first pass, the expression $\phi(\langle i_0 \rangle, \top)$ will be simplified to $\langle i_0 \rangle$, and j_1 will be given the value number $\langle i_0 \rangle$. Next, the expression $\langle i_0 \rangle + 1$ can be simplified to 2; it will be found in the *optimistic* table with value number $\langle i_2 \rangle$. During the second pass, the expression $\phi(\langle i_0 \rangle, \langle i_2 \rangle)$ will be found in the *optimistic* table with value number $\langle i_1 \rangle$, and $\langle i_1 \rangle + 1$ will be found with value number $\langle i_2 \rangle$. At this point the value numbers have stabilized, so we process the SCC using the *valid* table. Since entries already exist mapping $\phi(\langle i_0 \rangle, \langle i_2 \rangle)$ to $\langle i_1 \rangle$ and $\langle i_1 \rangle + 1$ to $\langle i_2 \rangle$, no new entries will be added to the *valid* table. Thus, the algorithm has determined that $i_1 \cong j_1$ and $i_2 \cong j_2$.

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 Figure 4.8, entries for $\langle i_1 \rangle$ and $\langle i_2 \rangle$ were added to both the *optimistic* and *valid* tables. On the other hand, the entry mapping the constant 2 to $\langle i_2 \rangle$ 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 j_1 and j_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 $\langle i_2 \rangle$; instead we add an entry to the *valid* table mapping 2 to a different name.

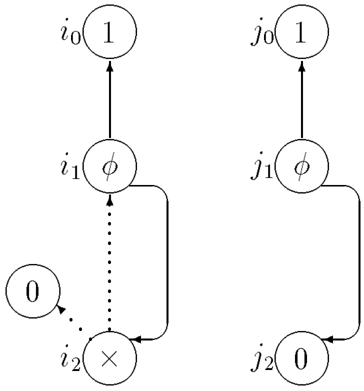
Recall that after the iteration stabilizes, we make one additional pass over the SCC using the *valid* table. The need to place expressions in both tables arises from constant folding and algebraic simplification. These transformations eliminate edges from the SSA graph. Thus, an SCC can be transformed 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. The example in Figure 4.9 illustrates how this might happen. The dashed edges in the SSA graph will be broken while iteratively analyzing the SCC containing i_1 and i_2. When the iteration stabilizes, the algorithm discovers that $i_2 = 0$. However, this analysis was performed using the *optimistic* table, where there is no entry mapping 0 to $\langle j_2 \rangle$. This entry is only in the *valid* table. Similarly, the entry mapping $\phi(\langle j_0 \rangle, \langle j_2 \rangle)$ to $\langle j_1 \rangle$ exists only in the *valid* table. Therefore, our algorithm has not yet discovered that $i_1 \cong j_1$ and $i_2 \cong j_2$. The additional pass over i_1 and i_2 using the *valid* table enables SCC-based value numbering to discover these two congruences.

4.7 Uninitialized Values

The handling of uninitialized values during value numbering is a complicated issue. Our implementation of SSA construction will remove an operation if it references an operand that is uninitialized on all paths. However, it is possible that a value could be initialized along some paths but not others. If this is the case, a special value, $\mathcal{U}$, will represent an uninitialized input to a ϕ-node. Any value numbering algorithm must correctly handle $\mathcal{U}$ when processing the routine. Our initial assumption was that since the behavior of an uninitialized value is undefined, the compiler is free to do just about anything. Therefore, we considered $\mathcal{U}$ to be congruent to every other value (just like $\top$). Further study showed that this assumption may not be correct. Whether or not it is correct depends on the source language definition. The example in Figure 4.10 demonstrates that this assumption can violate the "principle of least astonishment". In this example, the variable j is used to record the last

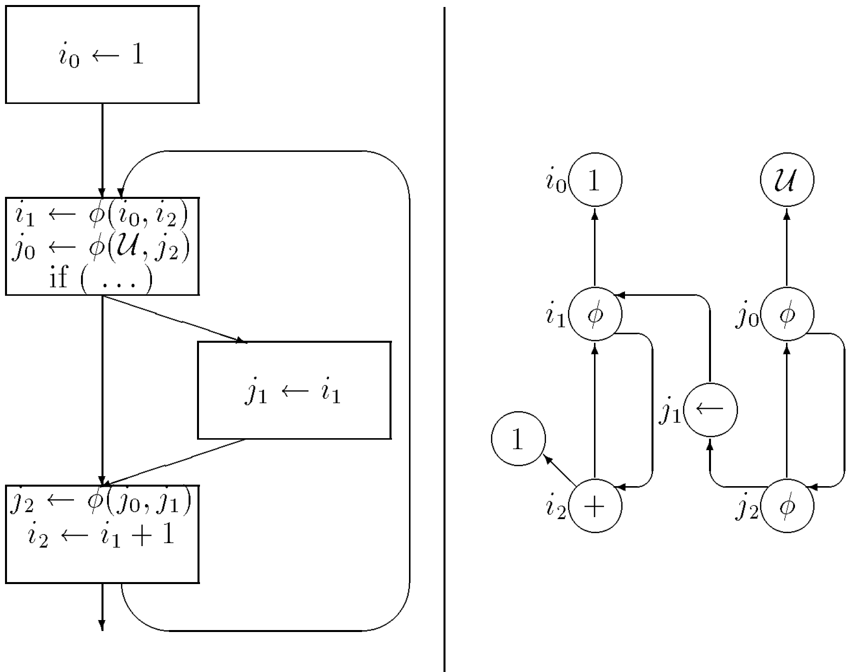
$i_0 \leftarrow 1$
$j_0 \leftarrow 1$
while $(\dots)$
    $i_1 \leftarrow \phi(i_0, i_2)$
    $j_1 \leftarrow \phi(j_0, j_2)$
    $i_2 \leftarrow i_1 \times 0$
    $j_2 \leftarrow 0$

SSA Form



SSA Graph

Figure 4.9 Example with edges removed from SCC



Control-flow Graph **SSA Graph**

Figure 4.10 Example with uninitialized values iteration on which some condition is true. If the condition is never true, the value of j remains uninitialized on exit from the loop. Since the value of j is not initialized before entering the loop, the corresponding parameter in the ϕ-node for j_0 is $\mathcal{U}$. If we assume that $\mathcal{U}$ is congruent to everything, SCC-based value numbering will go on to prove that $j_2 \cong i_1$. In other words, the value of j is now the last iteration of the loop, independent of the condition. We correct this problem by simply changing our assumptions about $\mathcal{U}$. Instead of treating $\mathcal{U}$ like $\top$, it must be treated as a distinct value number that is not congruent to anything. If we do this in our example, the algorithm will prove that $j_2 \ncong i_1$.

4.8 Summary

This chapter presents an original algorithm for value numbering that combines the advantages of hash-based value numbering and value partitioning. It is easy to understand and to implement, it can handle constant folding and algebraic identities, and it is global. We call the algorithm SCC-based value numbering because it is centered around the strongly connected components of the SSA graph. It runs in $\mathbf{O}(N \times D(\mathrm{SSA}))$ time, where N is the number of nodes, and $D(\mathrm{SSA})$ is the loop connectedness of the SSA graph. The loop connectedness can be as large as $N - 1$, but it is believed that, in practice, $D(\mathrm{SSA})$ is bound by a small constant. The maximum number of iterations required for any routine in our test suite is four. We prove that the algorithm finds at least as many congruences as hash-based value numbering and value partitioning.

Chapter 5

Code Removal

Chapters 2, 3, and 4 explain how to renumber the registers and ϕ-nodes so that congruent values are given the same number.[13] However, renumbering alone will not improve the running time of the routine; we must also remove the redundant computations. This chapter will present two techniques for removing code from a routine:

Dominator-Based Removal The technique suggested by Alpern, Wegman, and Zadeck is to remove computations that are dominated by another definition of the same value number [4]. Figure 5.1 shows an example routine that we can improve with this method. Since the computation of z in block B_1 dominates the computation in block B_4, the second computation can be removed.

AVAIL-Based Removal The classical approach is to compute the set of available expressions (AVAIL) and to remove computations that are in the AVAIL set

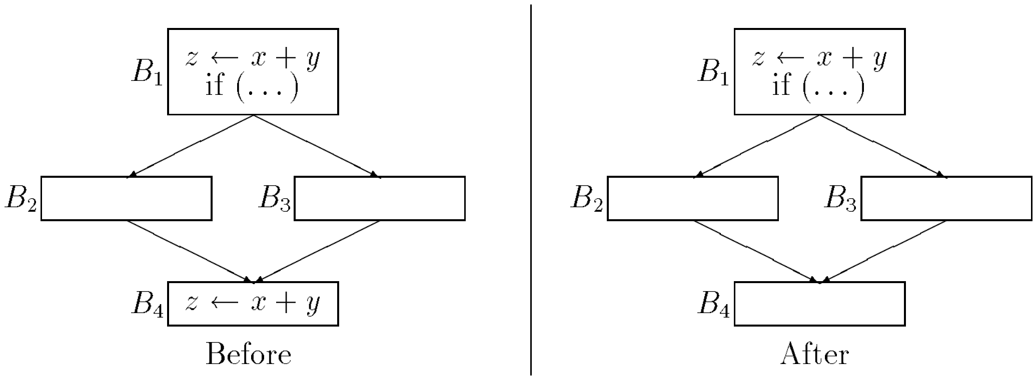


Figure 5.1 Program improved by dominator-based removal

[13]In Chapter 2, only the unified hash table algorithm will provide the consistent naming that we require.

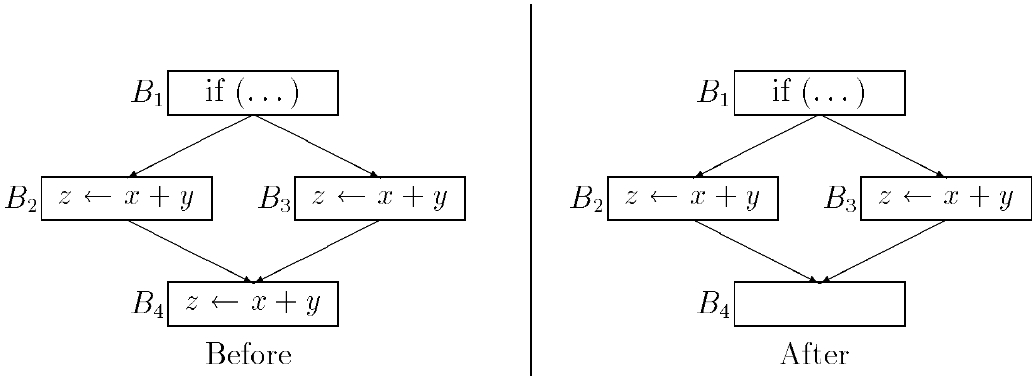


Figure 5.2 Program improved by AVAIL-based removal

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 z in Figure 5.1 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. However, there are improvements that can be made by the AVAIL-based technique that are not possible using dominators. Consider the routine in Figure 5.2. Since z is calculated in both B_2 and B_3, it is in the AVAIL set at B_4. Thus, the calculation of z in B_4 can be removed. However, since neither B_2 or B_3 dominate B_4, dominator-based removal could not improve this routine.

5.1 Dominator-Based Removal

To perform dominator-based removal, we consider each set of names with the same value number and look for pairs of members where one dominates the other. To make the algorithm efficient, we bucket sort the members of the set based on the preorder index in the dominator tree of the block where they are computed. A naive bucket sorting algorithm would keep a list of items defined for each block. Assume that items x and y are congruent and both are computed in block B. The bucket sorting algorithm would place them in a list indexed by the preorder index of B. See figure 5.3.

However, we can improve upon the naive algorithm. If more than one member is computed in the same block, only the one computed earliest in the block must be

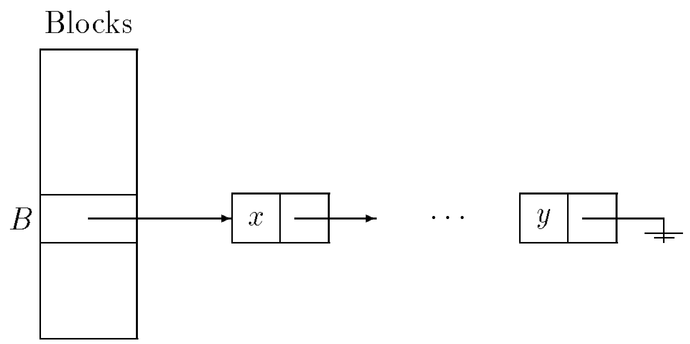


Figure 5.3 Naive bucket sorting algorithm

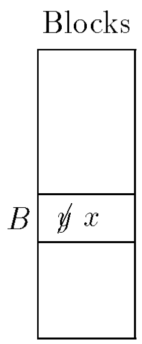


Figure 5.4 Better bucket sorting algorithm

considered, because it dominates all the others. Thus, instead of an array of lists, we can keep an array of SSA names. When an item is inserted into an entry that already contains an item, we can select the one that is computed earlier in the block. This item will be written into the array entry and the operation computing the other will be removed immediately. Assume that items x and y are congruent and both are computed in block B, and that x is computed earlier than y. The bucket sorting algorithm would replace y with x in the entry indexed by the preorder index of B. See Figure 5.4.

Once we have found the item computed earliest in each block, we can compare pairs of elements in the array 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 set.

During dominator-based removal, we'll need to perform an ancestor test on the dominator tree. To do this in constant time, we must know the preorder index and the number of descendants of each block in the dominator tree. We can decide if block b_1 dominates block b_2 by performing an ancestor test in the dominator tree. Let p_1 and p_2 be the preorder indices of b_1 and b_2 respectively, and let ND_1 be the number of descendants of b_1. Then b_1 dominates b_2 if and only if:

$$p_1 \leq p_2 < p_1 + ND_1$$

We use a "two-finger" algorithm for comparing items defined in different blocks. We consider adjacent pairs of non-zero entries in the `Blocks` array. The b_1 variable points to the first block of the pair in consideration. The b_2 variable points to the second block. These two pointers move through the `Blocks` array until each pair of adjacent blocks have been compared. For each pair, we check if b_1 dominates b_2. If so, we remove the instruction in the entry for block b_2; otherwise, we set b_1 to point to block b_2. The final step is to move b_2 to the next non-empty entry in the `Blocks` array.

5.2 AVAIL-Based Removal

The classical approach to redundancy elimination is to remove computations 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. An expression is available if it is computed along all paths from the beginning of the routine. If an operation computes a value already in the set of available expressions then it can be removed. First, we compute the AVAIL set for each block, then we remove any operations whose result is in the set.

Properties of the value numbered SSA form let us simplify the formulation of AVAIL. The traditional data-flow equations deal with the formal identity of lexical *names*, while our equations deal with identical *values* [14]. This is a very important distinction. We need not consider the killed set for a block because no values are redefined in SSA form, and value numbering preserves this property. Consider the code fragment in Figure 5.5. 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 be assigned different names. Since value numbering 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 equal to the definition of X that reaches the first assignment to Z.

The simplified data-flow equations are shown in Figure 5.6. Notice that the equation for AVOUT_i does not include a term for the expressions killed in block i. As we shall see in Chapter 6, the killed set requires a great deal of time to compute ($\mathbf{O}(N^2)$ time, where N is the number of value numbers). In our framework, we simply add the set of values defined in the block ($\mathrm{defined}_i$) to the set of values available at the beginning of the block (AVIN_i). The set $\mathrm{defined}_i$ is the set of value numbers with a definition in block i. This is a *superset* of the set of values generated in i (gen_i) which does not include any expressions whose definition is followed by a modification of one of its subexpressions. Further, $\mathrm{defined}_i$ is much easier to compute than gen_i.

Once we have computed AVAIL for each block, we are ready to remove instructions from the routine. We step through the blocks and use the AVIN set as a guide for

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

Figure 5.5 Example program

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

$$\mathrm{AVOUT}_i = \mathrm{AVIN}_i \cup \mathrm{defined}_i$$

Figure 5.6 Data-flow equations for AVAIL-based removal

removing instructions. We can remove any instruction whose result is in AVIN. If the instruction is not removed, we add its definition to the AVIN set because it is available to instructions later in the block.

5.3 Summary

This chapter presents two techniques for removing redundancies. Dominator-based removal eliminates computations that are dominated by another computation with the same value number. AVAIL-based removal is an improvement that relies on data-flow analysis to discover the set of available expressions and remove instructions whose value is in the set. We improve the data-flow framework by taking advantage of the properties of the value numbered SSA form. The improved framework deals with identical values rather than lexical names. It is simpler and faster to compute and it can remove more instructions than the traditional framework.

Chapter 6

Code Motion

The compiler can eliminate redundancies not only by removing computations but also by moving computations to less frequently executed locations. Many techniques rely on data-flow analysis to determine the set of locations where each computation will produce the same value and to select the ones that are expected to be least frequently executed.

6.1 Partial Redundancy Elimination

Partial redundancy elimination (PRE) is an optimization introduced by Morel and Renvoise that combines common subexpression elimination with loop invariant code motion [34, 21]. Partially redundant computations are redundant along some, but not necessarily all, execution paths. In general, PRE moves code upward in the routine to the earliest point where the computation would produce that same value without lengthening any path through the program. Notice that the computation of z in Figure 5.2 is redundant along all paths to block B_4, so it will be removed by PRE. On the other hand, the routine in Figure 6.1 cannot be improved using AVAIL-based

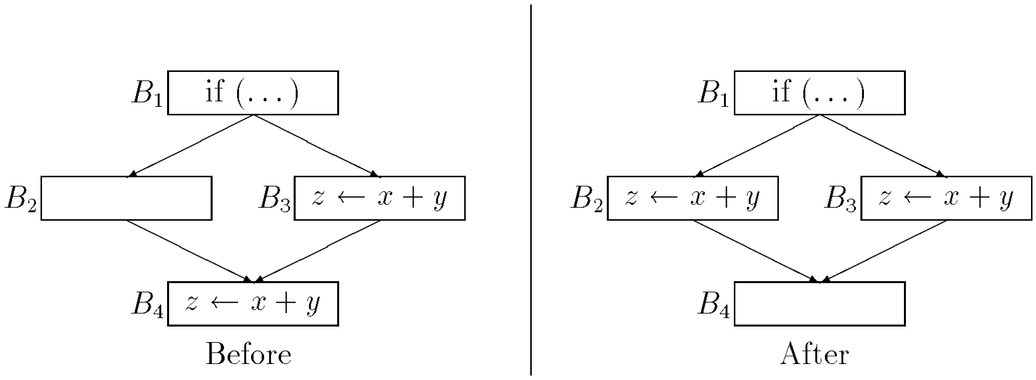


Figure 6.1 Program improved by partial redundancy elimination

removal because z is not available along the path through block B_2. The calculation 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.

6.2 Lazy Code Motion

Knoop, Rüthing, and Steffen describe a descendant of PRE, called lazy code motion (LCM) [29, 31]. Drechsler and Stadel present a variation of this technique that they claim is more practical [22]. The data-flow equations for this framework are shown in Figures 6.3 and 6.4. One advantage that Drechsler and Stadel's framework has over Knoop et al. is that it never inserts instructions in the middle of a block. Knoop et al. maintain an entry and an exit point for each expression in each block. This requires a substantial amount of memory, specifically $2 \times N \times B$ pointers to instructions, where N is the number of expressions and B is the number of blocks.

LCM avoids the unnecessary code motion inherent in PRE. This feature is important when code motion interacts with register allocation and other optimizations. 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 will study this issue further in Chapter 8.

Figure 6.2 gives an example of how unnecessary code motion can impact peephole optimization. The original routine shows a sequence of three instructions to perform a test and a branch. If PRE moves the first two instructions to each of the predecessor blocks, it has not changed the length of any path through the routine. However, notice the effect that this unnecessary code motion has on peephole optimization. This optimization can combine the `EQ` and the `BR` instructions to produce a `BReq` instruction if there is a single definition reaching the `BR`. The unnecessary code motion has resulted in two definitions that reach the `BR` instruction. Therefore, the instructions cannot be combined.

The common thread among all of these techniques is that they solve a sequence of data-flow equations that drive code motion. The first step of the analysis is to

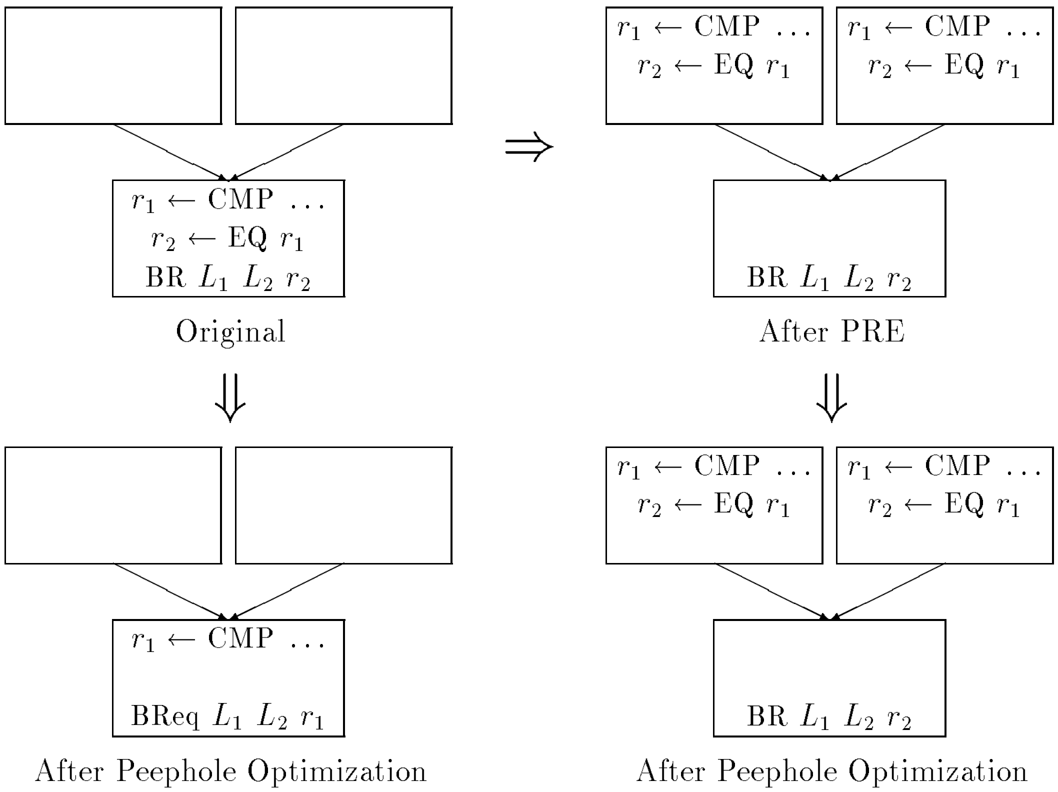


Figure 6.2 Unnecessary code motion

assign a unique number (bit-vector index) to each expression in the program. In our implementation, we enforce a naming scheme on the symbolic registers in the intermediate representation. The first rule is that multiple computations of the same register must compute the same expression, and all computations of the same expression must define the same register. In other words, each symbolic register defines exactly one lexical expression. This rule allows us to use the symbolic register name as the bit-vector index for each expression. Secondly, register numbers must be assigned bottom-up in each expression tree in increasing order. In other words, the register number defined by each expression must be higher than the register numbers of the operands. This rule allows us to simplify the analysis to determine which values depend on each other, and it allows us to insert more than one instruction in the same place in the correct order. Code that obeys these rules can be obtained in a number of ways. First, a carefully coded front end can produce intermediate code in this form by hashing each expression to determine the symbolic register that it must define. This approach may not be practical because it forces every front end to produce code in the proper form. Another drawback is that code motion can only be run once, and it must be the first optimization applied. A better approach is to enforce the naming scheme during value numbering. Hash-based value numbering will obey these rules if we simply assign value numbers in increasing order. Value partitioning and SCC-based value numbering require a separate pass over the code to enforce the naming scheme.

The next step is to determine which computations are eligible for motion and which are not. Morel and Renvoise refer to these as *expressions* and *variables*, respectively. Knoop et al. call them *terms* and *variables*. We will refer to them as *candidates* for code motion and *fixed values*. For each candidate, we must determine its subexpressions. This step is where we take advantage of the second rule in our naming scheme. We walk each expression tree bottom-up by simply considering each symbolic register in increasing order. At the point when we examine a symbolic register we have already determined the subexpressions for each of its operands. Therefore, the set of subexpressions for the current register is the union of the subexpressions for its operands. Once we have computed the subexpressions for each candidate, we are ready to compute the *dependences* for each fixed value. We say that a candidate, c, depends on fixed value, f, if f is in the set of subexpressions of c, $S[c]$. To compute dependences, we examine each $S[c]$. For each fixed value f in $S[c]$, we add c to the set of dependences for f. This process requires $\mathbf{O}(N^2)$ time, where N is the number of names.

Once we have computed the dependences for each fixed value, we are ready to compute the local predicates required for data-flow analysis. Each predicate is represented by a bit vector attached to each block. The first local predicate is *altered* – the set of candidates whose value is changed in the block.[14] This is the union of the dependences of every fixed value defined in the block. The *comp* predicate represents the set of candidates defined in the block that are not later altered. The *antloc* set contains the candidates that are defined before they are altered in the block. We compute all three predicates with a single pass over the block, examining each definition in order. If the definition is a fixed value, f, we add the set of dependences for f to the altered set and remove any dependences from the comp set. If the definition is a candidate, c, we add c to the comp set, and we add c to antloc if it is not in altered.

The data-flow analysis uses the local predicates to solve a series of equations (See Figures 6.3 and 6.4). The predicates used to modify the routine are $\mathrm{INSERT}_{i,j}$ for each edge (i, j) and DELETE_i for each block i. When inserting instructions, we take advantage of our second rule for naming symbolic registers – register numbers must be assigned bottom-up in each expression tree. We ensure that no instruction is placed before the definition of its operands by inserting the members of $\mathrm{INSERT}_{i,j}$ in increasing order.

6.3 Critical Edges

In reality, we want to avoid inserting instructions on edges whenever possible. Drechsler and Stadel point out that we can avoid this if *critical edges* have been split. A critical edge is an edge between a block with multiple successors and a block with multiple predecessors (*i.e.*, (i, j) is a critical edge if and only if $|\mathrm{succ}(i)| > 1$ and $|\mathrm{pred}(j)| > 1$). If edge (i, j) is the only predecessor of j, then $\mathrm{LATERIN}_j = \mathrm{LATER}_{i,j}$, so $\mathrm{INSERT}_{i,j} = \emptyset$. Further, if edge (i, j) is the only successor of i, then we can insert the candidates in $\mathrm{INSERT}_{i,j}$ at the end of block i. In general, we insert $\bigcap_{j \in \mathrm{succ}(i)} \mathrm{INSERT}_{i,j}$ at the end of block i rather than on the edge.

Critical edges can be removed by *splitting* – inserting an empty basic block along the edge. Figure 6.5 shows a critical edge and how it could be split. It is not always

[14] The analogous predicate in Drechsler and Stadel is $\overline{\mathrm{TRANSP}}$.

$$\mathrm{AVIN}_i = \begin{cases} \emptyset, & \text{if } i \text{ is the entry block} \\ \bigcap_{j \in \mathrm{pred}(i)} \mathrm{AVOUT}_j, & \text{otherwise} \end{cases}$$

$$\mathrm{AVOUT}_i = \mathrm{AVIN}_i - \mathrm{altered}_i \cup \mathrm{comp}_i$$

Availability

$$\mathrm{ANTOUT}_i = \begin{cases} \emptyset, & \text{if } i \text{ is the exit block} \\ \bigcap_{j \in \mathrm{succ}(i)} \mathrm{ANTIN}_j, & \text{otherwise} \end{cases}$$

$$\mathrm{ANTIN}_i = \mathrm{ANTOUT}_i - \mathrm{altered}_i \cup \mathrm{antloc}_i$$

Anticipatability

$$\mathrm{EARLIEST}_{i,j} = \begin{cases} \mathrm{ANTIN}_j \cap \overline{\mathrm{AVOUT}_i}, & \text{if } i \text{ is the exit block} \\ \mathrm{ANTIN}_j \cap \overline{\mathrm{AVOUT}_i} \cap & \\ \quad (\mathrm{altered}_i \cup \overline{\mathrm{ANTOUT}_i}), & \text{otherwise} \end{cases}$$

Earliest

Figure 6.3 Data-flow equations for lazy code motion – Part 1

$$\mathrm{LATERIN}_j = \begin{cases} \emptyset, & \text{if } j \text{ is the entry block} \\ \displaystyle\bigcap_{i \in \mathrm{pred}(j)} \mathrm{LATER}_{i,j}, & \text{otherwise} \end{cases}$$

$$\mathrm{LATER}_{i,j} = \mathrm{LATERIN}_i \cap \overline{\mathrm{ANTLOC}_i} \cup \mathrm{EARLIEST}_{i,j}$$

Later

$$\mathrm{INSERT}_{i,j} = \mathrm{LATER}_{i,j} \cap \overline{\mathrm{LATERIN}_j}$$

$$\mathrm{DELETE}_i = \begin{cases} \emptyset, & \text{if } i \text{ is the entry block} \\ \mathrm{ANTLOC}_i \cap \overline{\mathrm{LATERIN}_i}, & \text{otherwise} \end{cases}$$

Placement

Figure 6.4 Data-flow equations for lazy code motion – Part 2

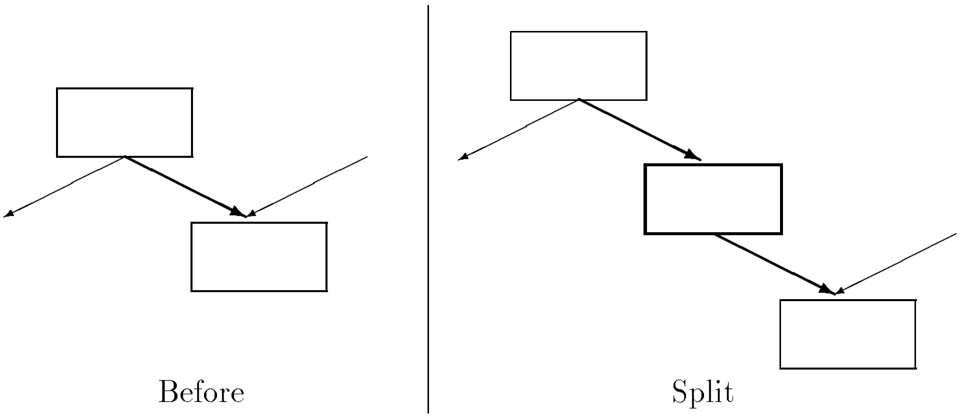


Figure 6.5 Splitting a critical edge

possible to split critical edges. If the predecessor ends in a "jump register" instruction, we cannot rewrite the code so that it will jump to the new block. This problem led us to look at the sources of jump-register instructions. In our compiler, they are generated for computed GOTO's in Fortran and for switch statements in C. In both cases, the code contains a specific sequence of instructions:

1. Compute the offset into a table of addresses
2. Load the address from the table
3. Jump to the location using a jump-register instruction

Given this sequence of instructions, we can split a critical edge if each jump-register instruction uses a different table and if we know which entry in the table corresponds to the critical edge. If so, we simply change the address in the table to point to the synthetic block. For this reason, we added a new opcode to our intermediate language, called "jump table", which tells the optimizer that the destination of the jump was loaded from a table. Therefore, the optimizer can split any critical edges leaving the block. This approach allows us to split all critical edges generated by C or Fortran. However, languages with first-class labels (*e.g.*, continuations in Scheme) can generate jump-register instructions that cannot be analyzed in this manner. In general, the inability to split all critical edges means that the code motion framework may be forced to place code on paths where it did not originally exist. However, the lazy code motion framework described by Knoop et al. can produce incorrect code for routines containing (unsplit) critical edges.

6.4 Moving LOAD Instructions

Significant improvements to the performance of optimized code can be gained by including LOAD instructions in the set of candidates. This is accomplished via the tags in our intermediate representation. Tags are described in detail in Chapter 2. The important feature of tags is that each instruction that might reference or define a memory location is labeled with the tag for that location. We extend the data-flow analysis to include tags as well as registers. We must increase the size of the bit vectors to hold both registers and tags, and we extend the analysis of subexpressions and dependences to include tags. Suppose that a particular LOAD instruction references tag t and defines register r. We consider t to be an operand of r, so r is in the set of dependences for t. During the local analysis, whenever a definition of t is encountered, r will be added to the altered set for the block. Therefore, the LOAD instruction cannot move past any definition of its tag. Note that the register operands (controlling the address) of the LOAD instruction will also inhibit its movement. Thus, a LOAD whose address varies with each iteration of a loop cannot be moved out of a loop even if the tag is not modified inside the loop.

We would like to allow STORE instructions to move as well. However, this extension is not as straightforward as it might seem. The problem lies in the fact that anti-dependences are not explicitly represented by tags.[15] The example in Figure 6.6 shows an anti-dependence between the reference of $A(i)$ and the definition of $A(1)$. If this anti-dependence is ignored, the definition of $A(1)$ will be incorrectly moved outside the loop because the address is loop invariant and the location is not written inside the loop. However, this motion is clearly unsafe because it will alter the value computed in the "sum" variable.

6.5 Summary

This chapter presents techniques for moving computations to less frequently executed locations. Partial redundancy elimination combines loop invariant code motion with common subexpression elimination. Partially redundant computations are redundant along some, but not necessarily all, execution paths. Lazy code motion is a descendant of partial redundancy elimination that avoids unnecessary code motion. We have extended these algorithms to allow motion of LOAD instructions.

sum $= 0.0$	sum $= 0.0$
do $i = 1, N$	$A(1) = \ldots$
sum $=$ sum $+ A(i)$	do $i = 1, N$
$A(1) = \ldots$	sum $=$ sum $+ A(i)$
enddo	enddo
Before	**Anti-dependence Ignored**

Figure 6.6 Incorrect motion of a STORE instruction

[15] An anti-dependence enforces the ordering between a read and a subsequent write of the same location.

Chapter 7

Value Driven Code Motion

Value-driven code motion (VDCM) is an improvement to classical code motion techniques that takes advantage of the results of global value numbering. Traditional data-flow analysis frameworks must assume that every definition produces a distinct value. Therefore, an instruction cannot move past a definition of one of its subexpressions. This restriction can be relaxed when certain definitions are known to produce redundant values. This information is discovered during value numbering, but previous techniques for code motion have not exploited it. By understanding how code motion interacts with global value numbering, we can simplify and improve the code motion framework at the price of constraining the order of the optimizations. Our approach is to modify the data-flow framework to account for the assumption that each definition represents a value rather than a lexical name. This approach can be applied to a variety of data-flow frameworks. In particular, this chapter focuses on lazy code motion presented in Chapter 6. That algorithm is provably optimal; this chapter shows that by changing our assumptions about the shape of the input program, we can produce a technique that both eliminates more redundancies and runs more efficiently.

We can prepare the routine for VDCM using any of the value numbering algorithms described in Chapters 2, 3, or 4.[16] In general, each algorithm discovers a different set of equivalences. The important feature that these algorithms share is that they can rewrite the names in the entire routine consistently to reflect the equivalences discovered.

The ability of the compiler to perform code motion is influenced heavily by the "shape" of the input program. Briggs and Cooper showed that global reassociation followed by value partitioning will transform code into a form that makes PRE more effective [5]. Further improvements are still possible. The focus of this chapter is to

[16] In Chapter 2, only the unified hash table algorithm will provide the consistent naming that we require.

extend the data-flow framework to operate on value equivalences rather than lexical names, just as we extended the framework for available expressions in Section 5.2. Because values are never killed, a computation of an expression can move across a definition of one of its operands if the value of that operand is available at the point where the computation is placed. In other words, a computation can be placed anywhere that the values of its operands are available.

7.1 VDCM Algorithm

The first step of VDCM is to find available expressions as described in Section 5.2. The only local predicate required for this framework is $\mathit{defined}_b$ – the set of values defined in block b. Using the results of the available expressions calculation, we can compute the other predicates needed for the remaining data-flow frameworks. These predicates take on a different meaning under VDCM because we are dealing with values rather than lexical names.

The predicate $\mathit{altered}_b$ represents the set of values that cannot move past some definition in block b. In our framework an instruction can move past the definition of one of its operands. However, an instruction cannot move to a point where the values of its operands cannot be made available. The concept of *ready* is defined recursively for each value v and each program point p. We say that v is ready if its fixed operands are available and its candidate operands are ready at p. Given the set of available values at point p, we can compute the set of ready values as follows: Initially, the set of ready values is the set of available values, then we traverse each expression tree bottom up and add any expression with all of its operands in the set. Finally, the set of values *altered* in block b is simply the set of values that are ready at the end of b but not at the beginning. In other words, a value is altered in block b if at least one of its subexpressions is computed in block b for the first time along some path in the CFG.

The algorithm for computing *altered* is shown in Figure 7.1. Intuitively, $\mathit{altered}_b$ for block b is derived by first computing the set of ready values at the beginning and end of the block, and then finding the difference. In practice, $\mathit{altered}_b$ can be computed more efficiently by starting with $\mathrm{AVOUT}_b - \mathrm{AVIN}_b$, then traversing each expression tree bottom up and adding any expression with one of its operands in the set. This requires $\mathbf{O}(N)$ time, where N is the number of names.

for each block b
    $altered_b \leftarrow \text{AVOUT}_b - \text{AVIN}_b$
    for each expression e in bottom-up order
        for each operand o of e
            if $o \in altered_b$
                $altered_b \leftarrow altered_b \cup \{e\}$

Figure 7.1 Algorithm for computing *altered* using values

for each expression e in bottom-up order
    for each operand o of e
        $S[e] \leftarrow S[e] \cup S[o] \cup o$
    for each fixed value $f \in S[e]$
        $D[f] \leftarrow D[f] \cup \{e\}$
for each block b
    for each definition x in b
        if x is a fixed value
            $altered_b \leftarrow altered_b \cup D[x]$

Figure 7.2 Algorithm for computing *altered* using lexical names

Compare this approach with the technique for computing the *altered* set using lexical names, shown in Figure 7.2. First, the subexpressions for each candidate and the set of dependences for each fixed value must be computed. To compute subexpressions, the analyzer must traverse each expression tree bottom up; for each expression, it computes the union of the subexpressions of its operands. To compute the set of dependences, it examines the subexpressions, $S[e]$ for each expression e. For each fixed value $f \in S[e]$, it adds e to the set of dependences for f. This process requires $\mathbf{O}(N^2)$ time, where N is the number of expressions. Finally, for each block b, the analyzer computes $altered_b$ as the union of the dependences for any fixed value defined in b.

To clarify the differences between the algorithms for computing *altered*, consider the expression tree in Figure 7.3. First, consider the value-driven algorithm. We will assume that the fixed value a is in $\text{AVOUT}_b - \text{AVIN}_b$. The algorithm will visit each expression in bottom-up order. The expression e_1 will be considered first; since one of its operands (a) is in the *altered* set, e_1 will be added. Since none of the operands

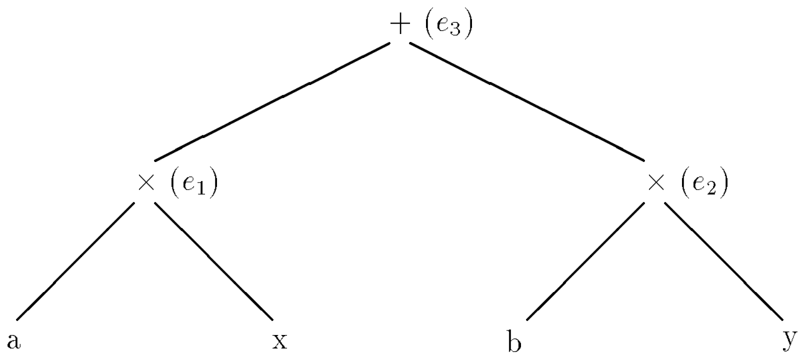


Figure 7.3 Expression tree

of e_2 are in *altered*, e_2 will not be added. Finally, e_3 will be added to *altered* because e_1 is a member of the set. Therefore, *altered* $= \{e_1, e_3\}$.

Now we will apply the algorithm based on lexical names to the expression tree in Figure 7.3. The first step is to compute the subexpressions for each expression in bottom-up order. The set of subexpressions for e_1, $S[e_1]$, is $\{a, x\}$. Similarly, $S[e_2] = \{b, y\}$, and finally $S[e_3] = \{e_1, a, x, e_2, b, y\}$. Notice that we have performed N bit-vector operations, each of size N, where N is the number of expressions. Therefore, computing subexpressions requires $\mathbf{O}(N^2)$ time. The next step is to compute the set of dependences for each fixed value. Since a is a member of $S[e_1]$ and $S[e_2]$, the dependences for a, $D[a]$, will be $\{e_1, e_3\}$. Similarly, $D[x] = \{e_1, e_3\}$, $D[b] = \{e_2, e_3\}$, and $D[y] = \{e_2, e_3\}$. Finally, we examine each definition in the block. When we encounter a definition of a, we add $D[a]$ ($\{e_1, e_3\}$) to the *altered* set.

The other predicate needed for each block b is the set of expressions that are locally anticipatable, antloc$_b$. Under the traditional framework, this is the set of expressions e computed in b before any of e's operands are modified in b. However, under the value-driven framework, antloc$_b$ is the set of values computed in b whose definition could legally be placed at the beginning of b. Any value computed in b can be computed at the beginning of b if that value is not altered in b. Therefore, we define antloc$_b$ as the set of values that are computed but not altered in b (*i.e.*, *defined*$_b$ − *altered*$_b$).

Given the value-driven versions of available expressions, *altered*, and locally anticipatable, the code motion proceeds as before. Specifically, we compute the set of values to insert on each edge (*i.e.*, $\mathrm{INSERT}_{i,j}$ for each edge $e = (i, j)$) and the set of values to delete from each block (*i.e.*, DELETE_b for each block b).

7.2 Examples

The example in Figure 7.4 demonstrates how VDCM is more powerful than LCM. The traditional framework would compute the set of subexpressions for r_1 as $\{x\}$, and for r_2 as $\{r_1, x\}$. Therefore, the set of dependences for x would be $\{r_1, r_2\}$, and the STORE to x in block B_2 must be assumed to alter the values of both r_1 and r_2 (*i.e.*, $altered_{B_2} = \{r_1, r_2\}$). However, the value numbering has discovered that the value of r_2 depends only on the value of r_1, and it is independent of the value stored into x inside the loop. In other words, r_2 must be the absolute value of the value of x on entry to the loop.

We will now explain how value-driven code motion would analyze this example. The first step is to compute available expressions using the equations in Figure 5.6. The solution to these equations shows that r_1 is available on entry and exit for block B_2 ($r_1 \in \mathrm{AVIN}_{B_2}$ and $r_1 \in \mathrm{AVOUT}_{B_2}$). We initialize the set $altered_{B_2}$ with $\mathrm{AVOUT}_{B_2} - \mathrm{AVIN}_{B_2}$. Since $r_1 \notin \mathrm{AVOUT}_{B_2} - \mathrm{AVIN}_{B_2}$, $r_2 \notin altered_{B_2}$, and VDCM is able to move the definition of r_2 outside the loop. On the other hand, LCM must leave the definition inside the loop.

Figure 7.5 demonstrates another example where VDCM is more powerful than LCM. Notice that the code stores the value of r_1 into $A(i)$ and later loads the value back into a register without changing the value of i. Value numbering has determined that the LOAD will produce the same value as r_1. Therefore, VDCM will remove the LOAD instruction. On the other hand, LCM must assume that the STORE affects the value of the LOAD, so it cannot remove it.

7.3 Summary

This chapter presents a new approach to data-flow analysis that takes advantage of facts discovered during value numbering. Traditional data-flow analysis frameworks operate on lexical names while our framework uses values. We apply this important distinction to the framework for lazy code motion. Despite the fact that lazy code motion is provably optimal, we can eliminate more redundancies using our technique.

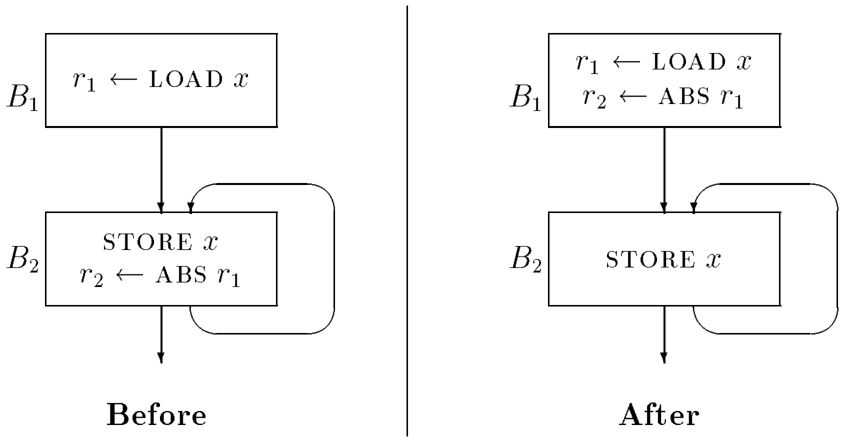


Figure 7.4 VDCM example

$$r_1 \leftarrow \ldots$$
$$\text{STORE } A(i)\ r_1$$
$$r_1 \leftarrow \text{LOAD } A(i)$$

Figure 7.5 Another VDCM example

Further, our algorithm runs faster than lazy code motion because the computation of the altered set for each block is greatly simplified.

Chapter 8

Relief of Register Pressure

This chapter will discuss the use of redundancy elimination techniques to relieve *register pressure* – the demand for registers at any point in a routine. It has been argued that eliminating redundancies will always increase register pressure [29, 31]. However, it is also possible that eliminating redundancies can reduce register pressure. The example in Figure 8.1 demonstrates how this can happen. When the second computation of x is removed, the lifetime of the first computation of x is lengthened. On the other hand, the lifetimes of y and z are shortened because their last use is now earlier in the program. The net effect is to reduce the number of live registers by one at the point marked by the "$\Rightarrow$". There are two other issues that further complicate the situation.

1. Increasing register pressure does not affect performance unless the pressure becomes greater than the number of physical registers on the target machine.
2. Other optimizations that run after redundancy elimination can change the register pressure in unpredictable ways.

$x \leftarrow y + z$	$x \leftarrow y + z$
$\leftarrow x$	$\leftarrow x$
$\Rightarrow$ y and z are live here	$\Rightarrow$ only x is live here!
$x \leftarrow y + z$	
$\leftarrow x$	$\leftarrow x$
Before	**After**

Figure 8.1 Example where redundancy elimination decreases register pressure

Routine	*Before Optimization*	*Before Redundancy Elimination*	*After Redundancy Elimination*	*After Optimization*
fpppp	137	1167	375	378
twldrv	175	212	311	322
deseco	36	91	136	172
supp	146	149	131	131
subb	125	126	125	125
tomcatv	48	39	80	118
efill	69	76	103	114
prophy	28	35	85	98
saturr	100	96	97	97
ddeflu	55	58	98	90
iniset	8	8	16	87
bilan	14	23	54	81
debflu	33	40	80	76
debico	24	52	48	73
repvid	22	26	57	69
inisla	20	31	60	64
paroi	18	23	53	61
bilsla	15	17	48	55
pastem	20	20	44	54
dyeh	37	40	50	51
drepvi	34	39	55	49
colbur	32	44	47	40
ihbtr	17	20	33	39
yeh	33	49	42	39
inithx	15	22	44	38
integr	12	16	35	38
cardeb	17	17	40	37
orgpar	27	36	36	34
gamgen	18	24	42	32
heat	23	24	31	29
inter	12	14	16	12
svd	35	35	52	58
decomp	22	22	35	49
rkfs	35	38	52	48
spline	14	17	34	37
fehl	22	25	37	34
fmin	24	22	32	30

Table 8.1 Register pressure at various points

Table 8.1 shows the register pressure at various points during optimization for the routines in each benchmark suite in which the register pressure is above 32.[17] Each entry in the table represents the maximum register pressure at any point in the routine. In almost all cases, redundancy elimination increased register pressure. However, in only 40% of the routines did the register pressure cross the 32 register threshold during the redundancy elimination phase. The register pressure in 5 of the routines is reduced by eliminating redundancies. It is reduced significantly for `fpppp`. Finally, the later optimization passes change the register pressure in unpredictable ways – usually the pressure is raised; it is lowered in 35% of the routines.

One approach to relief of register pressure might be to somehow limit redundancy elimination so that it does not raise the register pressure above the number of physical registers. However, this would still not account for the increase in register pressure caused by later optimizations. Therefore, we feel that the most effective way to relieve register pressure will be a transformation that runs immediately before register allocation. Another advantage of this approach is that it does not depend on the algorithm used for redundancy elimination. The idea is to insert instructions that reduce the register pressure but are cheaper than the LOAD and STORE operations that will be inserted during register allocation [11, 7]. Further, these instructions will be placed on a more global basis than the spill code. In some sense, this transformation will undo the adverse effects of redundancy elimination. This transformation will operate as follows:

1. Perform value numbering to identify expressions with the same value.

2. Identify locations in the routine where the pressure is greater than $\mathcal{R}$ – the number of physical registers on the target machine. This is accomplished using traditional live variable analysis. The register pressure at each point in the routine is estimated by the size of the "live" set. The true register pressure in the block may change as a result of the copy coalescing that occurs during register allocation.

3. Heuristically identify calculations that will decrease the pressure. For each block where the register pressure is too high, we search for expressions whose result and operands are live at the beginning and end of the block but not mentioned

[17] We consider 32 to be a good estimate of the number of physical registers on a typical machine.

inside the block. Compile-time constants are particularly attractive because they have no register operands. If such a calculation were placed at the end of the block, the register pressure inside the block would decrease by one.

4. Select the location where inserting these calculations will result in the least run-time impact. We will place instructions as late as possible without crossing a use of the result or lengthening any path at a greater nesting depth.

5. Apply dead code elimination to remove any definitions that have been "masked" by inserted instructions.

8.1 Identifying Blocks with High Register Pressure

We locate regions of high register pressure using data-flow analysis to identify live registers. For each block, we begin with the LIVEOUT set and examine the instructions in reverse order. At each instruction, we remove any defined registers from the set and insert any used registers. The register pressure is simply the size of the set at any point. We record the maximum register pressure for any point inside each block.

8.2 Choosing Expressions to Relieve Pressure

Once we have identified the blocks with high register pressure, we are ready to select expressions to relieve the pressure. We begin with the set of expressions whose result is live across the block and whose result and operands are not touched within the block. For each expression in this set, we must also ensure that its operands are live on exit to the block. Otherwise, inserting the expression at the end of the block would shorten the live range of the result while lengthening the live range of one or more of the operands. The net result would be unchanged or even increased register pressure.

Once we have narrowed down the choice of expressions, we heuristically choose one expression at a time until the register pressure for the block drops to $\mathcal{R}$ or the set becomes empty. We have identified several heuristics.

Heuristic 0 Choose expressions in any order.

Heuristic 1 Give priority to registers constrained in predecessors. The example in Figure 8.2 motivates this heuristic. It shows three consecutive blocks with high register pressure. If three different expressions are chosen to relieve pressure,

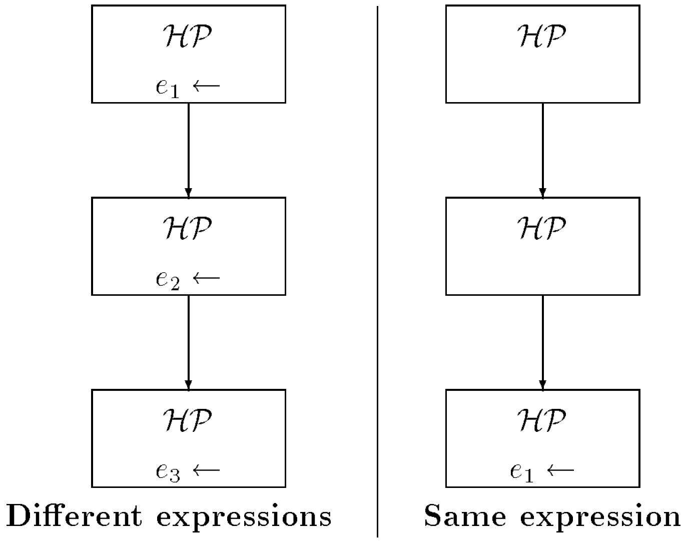


Figure 8.2 Motivating example for heuristic 1

then three instructions will be needed. However, if the same expression is chosen for each block, only one instruction will be needed. During the next step, the correct location for the calculation will be chosed at the end of the third block.

Heuristic 2 Allow existing definitions to participate in code motion. Figure 8.3 shows how inserting an expression at the end of a block with high pressure can render another definition of the expression partially dead. If we treat both definitions as candidates for code motion, we will eliminate the partially dead computation.

Heuristic 3 Limit code motion to expressions actually inserted. Figure 8.4 shows how heuristic 2 can move an expression into an empty basic block that would otherwise have been removed by a later optimization pass. The result is that the instruction is not executed on the final iteration of the loop at the cost of executing an extra jump on every iteration. This problem occurs in the `gamgen` routine. For this reason, it is often useful to limit the expressions to be moved to only those involved in relieving pressure.

Heuristic 4 Give priority to compile time constants. Because these expressions have no register operands, we can insert them without worrying about lengthening the live range of the operands. This heuristic is similar to the rematerialization

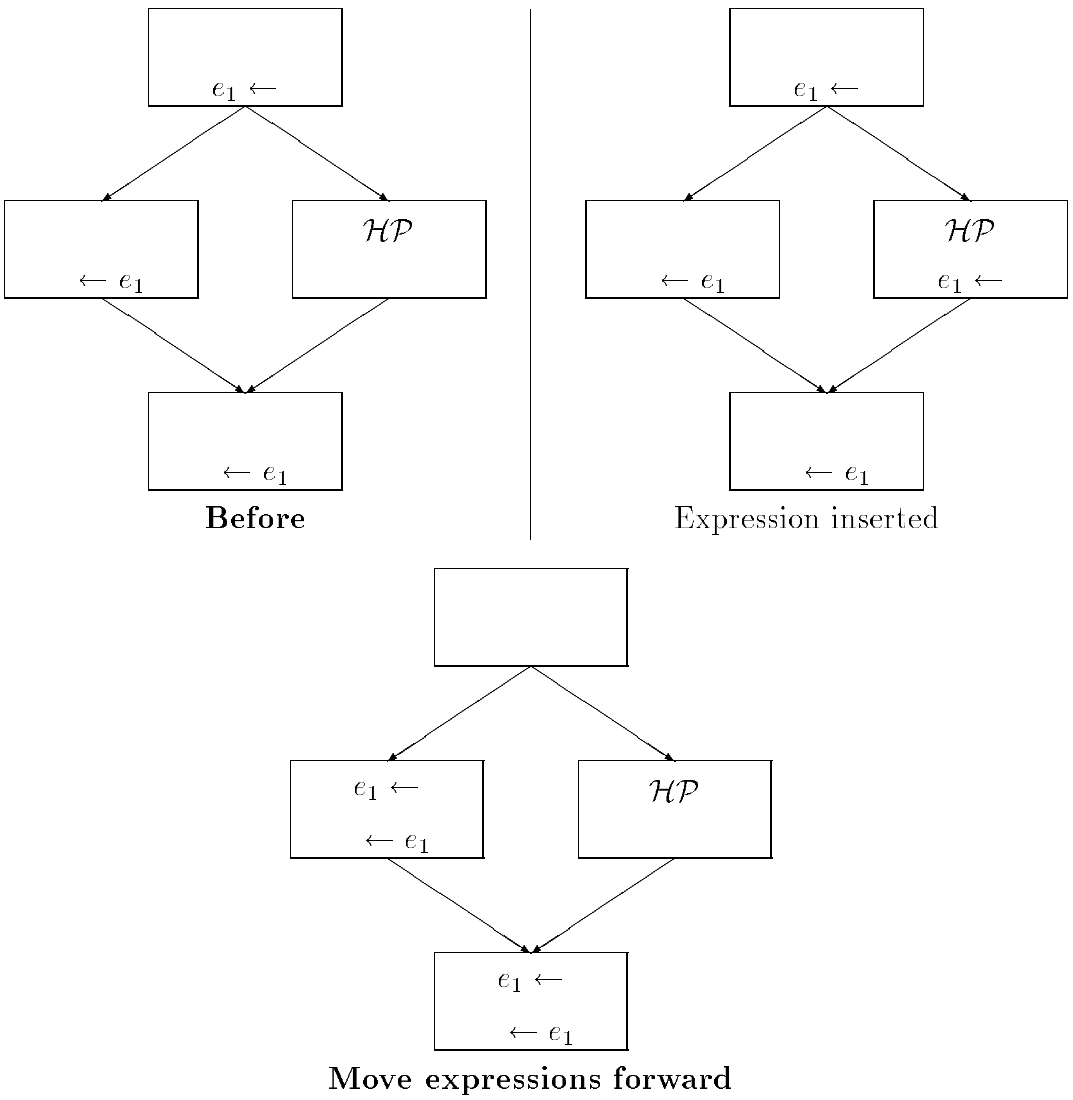


Figure 8.3 Motivating example for heuristic 2

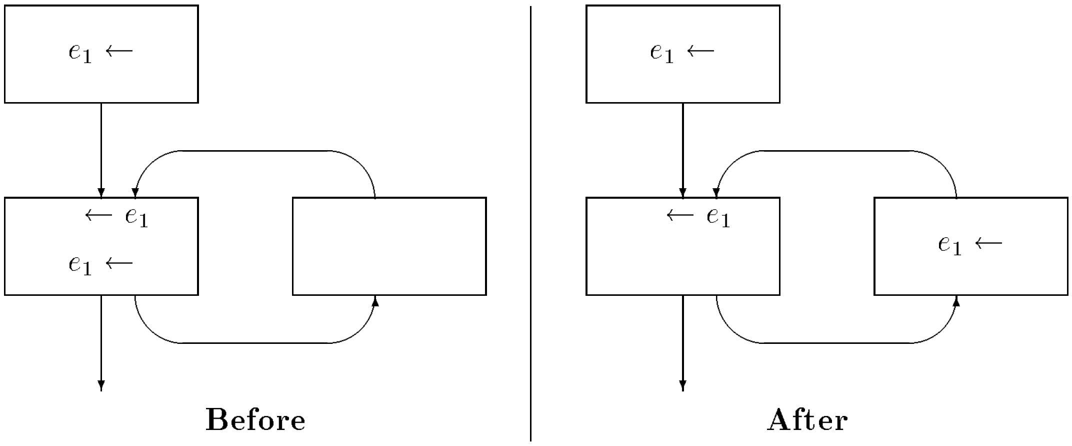


Figure 8.4 Motivating example for heuristic 3

technique of Briggs, Cooper, and Torczon [6]. The key difference is where the code to compute the constant is inserted. Rematerialization is attempted for each live range that is spilled during register allocation; any use within the live range that is shown to be a constant will be preceded by a LOAD-IMMEDIATE instruction rather than a LOAD. If all uses are handled in this fashion, then no STORE instructions are needed at the definitions within the live range. In contrast, this heuristic inserts a LOAD-IMMEDIATE instruction at the end of a block with high pressure and moves that instruction forward based on the analysis described in the next section.

Heuristic 5 Give priority based on nesting depth. Since expressions are moved as close as possible to their uses, we look for expressions whose uses are at the outermost nesting depth.

8.3 Selecting Locations to Insert Instructions

We select locations to insert each expression using data-flow analysis similar to the second phase of lazy code motion (See Chapter 6). We move each expression as late as possible subject to the following:

1. Do not pass a use of the expression. Passing a use would force that use to obtain its value from a definition before the block where we are attempting to relieve pressure.

$$\mathrm{RELIEFIN}_i = \begin{cases} \emptyset, & \text{if } i \text{ is the entry block} \\ \displaystyle\bigcup_{\substack{j \in \mathrm{pred}(i) \\ \mathrm{depth}(i) \leq \mathrm{depth}(j)}} \mathrm{RELIEFOUT}_j, & \text{otherwise} \end{cases}$$

$$\mathrm{RELIEFOUT}_i = \mathrm{RELIEFIN}_i - \mathrm{exposed}_i \cup \mathrm{relief}_i$$

Relief

$$\mathrm{INSERT}_{i,j} = (\mathrm{RELIEFOUT}_i - \mathrm{RELIEFIN}_j) \cap \mathrm{LIVEIN}_j$$

Insert

Figure 8.5 Data-flow equations for relief of register pressure

2. Do not lengthen any path at a greater nesting depth. The restriction of not lengthening *any* path can prohibit conditionally executed expressions from moving out of loops. On the other hand, allowing any path to be lengthened can result in expressions moving into loops. Therefore, we have chosen to compromise.

The data-flow equations for selecting insertion points are given in Figure 8.5. The local predicates required for each block, i, are $\mathrm{exposed}_i$ and relief_i. They represent expressions with an exposed use in i and expressions initially inserted in i, respectively. Notice that we only consider the predecessors with an equal or greater nesting depth when computing RELIEFIN.

Recall that the data-flow framework for lazy code motion computes a set of expressions to insert on each edge of the CFG. In Section 6.3, we showed that inserting expressions on edges was not necessary if critical edges in the graph had been split. The same is true in this framework.

8.4 Results

Table 8.2 shows the register pressure for several routines using each of the heuristics described above. Since the heuristics can only relieve register pressure that exists across basic blocks, they are not effective on `fpppp`, which contains only one basic block. Also, the `gamgen`, `heat`, `inter`, and `fmin` routines are not affected because their register pressure after optimization is below the 32 register threshold. When the heuristics are applied, each one reduces the pressure significantly. However, the pressure in only 21% of the routines was lowered below its level achieved before redundancy elimination (see Table 8.1). Many of the heuristics reduce the maximum pressure to the same level. If the maximum pressure is high, this means that all possible expressions were chosen and there is simply no decision for the heuristics to make. However, this does not mean that each heuristic will change the routine in the same way. Recall that register pressure can vary throughout a routine and that even though there are blocks where the pressure cannot be brought below the threshold, there are other blocks where the pressure can be relieved. The differences between the heuristics become evident in blocks where the pressure is near the threshold. In these blocks, different heuristics will choose different expressions, and different behavior will be observed at run time. The register pressure inside the `iniset`, `ihbtr`, `inithx`, and `cardeb` routines was reduced to a point where it can be allocated on a 32 register machine without spill code. We are not able to achieve this goal on every routine simply because there are not enough expressions whose insertion would relieve pressure. Chapter 9 will present data showing the improvements in instruction counts that result from each of the heuristics.

8.5 Summary

We have presented a technique that can reduce register pressure by introducing redundancies into a routine. As a result, less spill code will be inserted during register allocation, and the running time of the routine is reduced. Intuitively, the technique proceeds by (1) identifying blocks with high register pressure in the routine, (2) heuristically choosing expressions that will reduce register pressure if inserted at the end of the block, and (3) moving definitions as close to their uses as possible.

One key aspect of this technique is that definitions cannot move past a use. This includes uses in register-to-register copy instructions. Since many of these instructions are removed during the coalescing phase of register allocation, further opportunities

Routine	*Without Relief*	*Heuristic 0*	*1*	*2*	*3*	*4*	*5*
fpppp	378	378	378	378	378	378	378
twldrv	322	269	269	269	269	269	269
deseco	172	146	146	146	146	146	146
supp	131	130	130	130	130	130	130
subb	125	125	125	125	125	125	125
tomcatv	118	82	82	82	82	82	82
efill	114	98	98	98	98	98	98
prophy	98	81	81	81	81	81	81
saturr	97	97	97	97	97	97	97
ddeflu	90	64	64	64	64	64	64
iniset	87	32	32	32	32	32	32
bilan	81	71	71	71	71	71	71
debflu	76	48	48	48	48	48	48
debico	73	67	67	67	67	67	67
repvid	69	56	56	56	56	56	56
inisla	64	62	62	62	62	62	62
paroi	61	47	47	47	47	47	47
bilsla	55	50	50	50	50	50	50
pastem	54	40	40	41	40	40	40
dyeh	51	51	51	51	51	51	51
drepvi	49	40	40	40	40	40	40
colbur	40	38	38	38	38	38	38
ihbtr	39	32	32	31	32	32	32
yeh	39	36	36	36	36	36	36
inithx	38	32	32	32	32	32	32
integr	38	34	34	36	34	34	34
cardeb	37	31	31	31	31	31	31
orgpar	34	34	34	33	34	34	34
svd	58	47	47	50	47	47	47
decomp	49	37	36	36	36	36	36
rkfs	48	42	42	42	42	42	42
spline	37	32	32	28	32	31	32
fehl	34	33	33	33	33	33	33

Table 8.2 Register pressure using relief heuristics

for movement may be created. This observation leads us to believe that this technique might be more effective as a part of the register allocator rather than a separate pass that runs before allocation. Such an allocator could insert expressions and perform forward motion following each application of the coalescing phase. Forward motion could not only be applied to expressions inserted during the current phase, but also to those expressions inserted in previous phases. Thus, any expression whose forward motion was blocked by a copy that was subsequently coalesced could be moved closer to its "true" uses.

Another advantage of incorporating this technique into the register allocator is that code motion could be applied to the spill code. Rather than inserting a STORE instruction after each definition in a live range and a LOAD instruction before each use, the allocator could identify regions of high pressure and insert a STORE at the beginning and a LOAD at the end. Then, the LOAD instructions could be moved forward as far as possible using the existing data-flow framework and the STORE instructions could be moved as early as possible using an analogous framework.

Chapter 9

Experimental Results

A theoretical comparison of the redundancy elimination techniques reveals that SCC-based value numbering and value-driven code motion are never worse than their competitors in terms of eliminating redundancies. However, an equally important question is how much this theoretical distinction matters in practice. Practical considerations force us to ask two key questions:

1. How often do opportunities for improved redundancy elimination arise?
2. How does improved redundancy elimination interact with other optimization passes?

To assess the real impact of these techniques, we have implemented all of the optimizations in our experimental Fortran compiler.

9.1 Experimental Compiler

We have built an experimental compiler 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.

Figure 9.1 shows a sample routine written in ILOC. Each basic block in the routine is identified by its label, and each block ends with an explicit flow-of-control operation. The routine begins with a FRAME operation that defines the registers passed from the calling routine. It ends with a RTN operation. Each operation is labeled with the a line number in the source code. Each operation consists of an opcode, a list of constants, a list of referenced registers, and a list of defined registers. Any of these lists can be empty. The "=>" symbol indicates the beginning of the list of defined registers. Constant integers appear with no prefix; memory locations appear with "@" as the first character, and registers appear with the letter "r" as the first character. All opcodes that define registers have a type (indicated by the first letter

```
_.bubble_:   10   FRAME    0 => r0 r1 r2 r3
             10   iSLDor   @n 4 0 r3 => r4
             0    iSLI     2 r4 => r5
             10   iLDI     1 => r6
             0    iADDI    -1 r4 => r7
             0    iSLI     2 r7 => r8
             10   iCMP     r7 r6 => r9
             10   BRlt     LL0005 LL0006 r9

LL0006:      0    iADDI    -4 r2 => r10
             0    iADD     r8 r10 => r11
             0    iADD     r5 r10 => r12
             0    iADDI    4 r10 => r24
             0    JMPl     LL0004

LL0004:      0    iADDI    4 r24 => r13
             13   iCMP     r12 r13 => r14
             13   BRlt     L4 LL0009 r14

L4:          0    i2i      r24 => r22
             0    JMPl     LL0008

LL0009:      0    i2i      r13 => r23
             0    i2i      r24 => r22
             0    JMPl     LL0007

LL0007:      14   iLDor    @*a 4 0 r23 => r15
             14   iLDor    @*a 4 0 r22 => r16
             14   iCMP     r15 r16 => r17
             14   BRlt     LL0010 LL0011 r17

LL0010:      0    i2i      r23 => r22
             0    JMPl     LL0011

LL0011:      0    iADDI    4 r23 => r23
             16   iCMP     r12 r23 => r18
             16   BRge     LL0007 LL0008 r18

LL0008:      18   iLDor    @*a 4 0 r24 => r19
             19   iLDor    @*a 4 0 r22 => r20
             19   iSTor    @*a 4 0 r24 r20
             20   iSTor    @*a 4 0 r22 r19
             22   iCMP     r11 r13 => r21
             22   BRge     L9 LL0005 r21

L9:          0    i2i      r13 => r24
             0    JMPl     LL0004

LL0005:      23   RTN      r0
```

Figure 9.1 Sample ILOC routine

of the opcode). The following data types are supported: byte, integer, floating point, double precision, complex, and double precision complex. In this example, all registers are integers. LOAD and STORE operations support two addressing modes. The *offset-register* mode (*e.g.*, iLDor) computes the address by adding a constant offset to the value in a register. The *register-register* mode (*e.g.*, iLDrr) computes the address by adding the values in two 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 C code instrumented to count the number of ILOC instructions executed. The number of instructions executed is a good (but not perfect) indication of the program's execution time. The following details are overlooked:

1. Not all instructions require the same number of cycles to execute.

2. The effects of a back end are not measured. In particular, the effects of register allocation and scheduling might change the results. Solving these problems globally requires a heuristic approximation to an NP-complete problem. The heuristic chosen must be tailored to work well with the code expected from the optimizer.

3. Machine features such as pipelines and memory hierarchies are not taken into account.

The common thread among these shortcomings is that they all depend on features of a specific target machine. Therefore, we feel that our experiments are useful in measuring the effectiveness of machine-independent optimizations.

9.2 Tests Performed

In this experiment, we optimized over 50 routines from the Spec benchmark suite and from Forsythe, Malcolm, and Moler's book on numerical methods [23]. We refer to the latter as the FMM benchmark. Each routine was optimized in several different ways by varying the type of value numbering and code removal/motion. Table 9.1 shows the possible combinations. The entries marked with a "∘" represent optimizations that existed prior to the beginning of this research; entries marked with a "•" represent contributions of this research.

To achieve accurate comparisons, we varied only the type of redundancy elimination performed. Routines were optimized with the sequence of global reassociation [5],

	Single Basic Blocks	Extended Basic Blocks	Dominator Based Removal	AVAIL Based Removal	Lazy Code Motion	Value-driven Code Motion
Hash-based	○	○	●	●	●	●
Partitioning			○	●	○	●
SCC			●	●	●	●

Table 9.1 Tests performed

redundancy elimination (different in each test), global constant propagation [45], operator strength reduction (see Appendix A), redundancy elimination[18], global constant propagation[18], global peephole optimization, dead code elimination [18, Section 7.1], copy coalescing, and a pass to eliminate empty basic blocks.

9.3 Raw Instruction Counts

Figures 9.2 through 9.7 present ILOC instruction counts for each combination in Table 9.1. Table 9.2 presents a guide for these figures. Remember that the instruction counts show the performance of the different techniques in the context of an optimizing compiler. Thus, even though one technique may remove more redundancies than another, subsequent optimization passes may negate this improvement.

Type of redundancy elimination	Spec	FMM
Hash-based, vary code removal/motion	Figure 9.2	Figure 9.5
Partitioning, vary code removal/motion	Figure 9.3	Figure 9.6
SCC-based, vary code removal/motion	Figure 9.4	Figure 9.7

Table 9.2 Key to raw instruction count figures

[18]These optimizations are repeated to clean up after strength reduction.

Hash Based						
	Single	Extended	Dominator	AVAIL	LCM	VDCM
tomcatv	391116123	324009765	321398525	321398525	227904139	226608739
twldrv	77380898	65204805	62457308	62394396	63462668	61658575
gamgen	102266	86685	86282	86282	86256	86256
iniset	84119	38115	38115	38115	38077	38077
deseco	17500	16271	15622	15608	14863	14823
debflu	5164	4855	4553	4553	4364	4330
prophy	5026	4342	3777	3757	3939	3939
pastem	4817	4078	3570	3570	3582	3576
fpppp	3925	3922	3922	3922	3990	3922
repvid	3858	3427	3074	3021	2859	2846
bilan	3670	3721	3419	4034	3452	3452
paroi	3591	3574	3495	3503	3724	3601
debico	3567	3393	3364	3335	3061	3061
inithx	3121	2874	2552	2552	2628	2620
integr	2479	2338	2124	2124	2444	2444
sgemv	1236	1042	1035	1035	791	791
cardeb	1132	1076	1029	1042	785	785
sgemm	1130	982	981	981	834	834
inideb	982	920	891	891	788	775
supp	896	812	693	693	693	693
saxpy	859	759	759	759	467	467
ddeflu	809	773	741	739	704	701
subb	630	539	539	539	539	539
fmtset	604	365	360	360	343	360
ihbtr	492	450	456	456	478	487
drepvi	372	329	272	268	267	267
x21y21	358	299	299	299	263	263
saturr	316	318	244	242	243	243
fmtgen	280	232	205	200	176	176
efill	270	251	216	213	215	215
si	246	177	164	164	165	165
heat	215	212	178	178	177	177
dcoera	171	153	144	144	155	144
lclear	165	127	127	127	91	91
orgpar	154	134	129	129	117	117
yeh	147	142	124	124	134	124
colbur	132	130	117	117	117	117
coeray	131	113	104	104	100	96
drigl	124	120	115	115	115	115
lissag	106	89	89	89	82	82
aclear	101	79	79	79	59	59
sortie	90	88	81	81	78	78
sigma	54	48	48	48	48	48
hmoy	27	23	23	23	23	23
dyeh	25	25	25	25	25	25
vgjyey	22	22	20	20	20	20
arret	19	19	19	19	19	19
inter	10	10	10	10	10	10
bilsla	6	6	6	6	6	6
inisla	6	6	6	6	6	6

Figure 9.2 Number of ILOC operations for hash-based value numbering techniques – Spec benchmark

Partitioning				
	Dominator	AVAIL	LCM	VDCM
tomcatv	429513447	429513447	228685858	227390458
twldrv	65643989	65581077	66494004	64689911
gamgen	152725	152725	137858	137858
iniset	38115	38115	38077	38077
deseco	15117	15066	14723	14710
debflu	4987	4987	4569	4535
prophy	3419	3447	4045	4045
pastem	3717	3717	3675	3669
fpppp	3988	3988	4056	3988
repvid	3215	3162	2915	2915
bilan	3560	3560	3544	3544
paroi	4147	4151	4145	4039
debico	3809	3627	3375	3375
inithx	2653	2653	2682	2675
integr	2124	2124	2444	2444
sgemv	1834	1834	1197	1197
cardeb	1540	1525	884	884
sgemm	1230	1230	838	838
inideb	1175	1175	916	916
supp	693	693	693	693
saxpy	1009	1009	472	472
ddeflu	762	761	726	725
subb	539	539	539	539
fmtset	416	416	354	371
ihbtr	465	465	475	484
drepvi	305	301	300	300
x21y21	299	299	263	263
saturr	244	242	243	243
fmtgen	214	215	180	180
efill	219	215	217	217
si	169	169	170	170
heat	181	181	180	180
dcoera	149	149	160	149
lclear	127	127	91	91
orgpar	132	132	118	118
yeh	124	124	134	124
colbur	129	129	129	129
coeray	109	109	105	101
drigl	129	129	129	129
lissag	102	102	89	89
aclear	79	79	59	59
sortie	85	85	82	82
sigma	48	48	48	48
hmoy	34	34	34	34
dyeh	26	26	26	26
vgjyey	20	20	20	20
arret	19	19	19	19
inter	13	13	13	13
bilsla	6	6	6	6
inisla	6	6	6	6

Figure 9.3 Number of ILOC operations for value partitioning techniques – Spec benchmark

SCC-Based				
	Dominator	AVAIL	LCM	VDCM
tomcatv	321393425	321393425	227899039	226603639
twldrv	63070603	63007691	64169355	62365262
gamgen	83855	83855	86256	86256
iniset	38115	38115	38077	38077
deseco	14018	14018	14462	14423
debflu	4553	4553	4364	4330
prophy	3297	3325	3923	3923
pastem	3544	3544	3566	3550
fpppp	3922	3922	3990	3922
repvid	2912	2859	2779	2766
bilan	3055	3055	3369	3369
paroi	3497	3505	3726	3603
debico	3070	3070	2963	2963
inithx	2552	2552	2627	2620
integr	2124	2124	2444	2444
sgemv	1035	1035	791	791
cardeb	1029	1042	785	785
sgemm	981	981	834	834
inideb	892	892	789	776
supp	693	693	693	693
saxpy	759	759	467	467
ddeflu	735	734	702	701
subb	539	539	539	539
fmtset	360	360	343	360
ihbtr	432	432	478	476
drepvi	272	268	267	267
x21y21	299	299	263	263
saturr	244	242	243	243
fmtgen	205	200	176	176
efill	216	213	215	215
si	164	164	165	165
heat	178	178	177	177
dcoera	144	144	155	144
lclear	127	127	91	91
orgpar	129	129	117	117
yeh	124	124	134	124
colbur	117	117	117	117
coeray	104	104	100	96
drigl	115	115	115	115
lissag	90	90	83	83
aclear	79	79	59	59
sortie	81	81	78	78
sigma	48	48	48	48
hmoy	23	23	23	23
dyeh	25	25	25	25
vgjyey	20	20	20	20
arret	19	19	19	19
inter	10	10	10	10
bilsla	6	6	6	6
inisla	6	6	6	6

Figure 9.4 Number of ILOC operations for SCC-based value numbering techniques – Spec benchmark

Hash-Based						
	Single	Extended	Dominator	AVAIL	LCM	VDCM
svd	6392	5502	5081	5059	4445	4281
fmin	2072	1138	941	941	880	881
zeroin	1418	912	836	836	748	748
spline	1054	937	928	928	895	886
decomp	847	724	714	707	658	657
fehl	744	708	704	704	646	646
rkfs	552	384	332	332	279	279
urand	226	225	223	223	225	223
solve	208	189	186	186	187	185
seval	121	113	78	78	77	77
rkf45	35	35	35	35	35	35

Figure 9.5 Number of ILOC operations for hash-based value numbering techniques – FMM benchmark

Partitioning				
	Dominator	AVAIL	LCM	VDCM
svd	5362	5340	4532	4384
fmin	941	941	880	881
zeroin	836	836	748	748
spline	972	972	924	915
decomp	713	706	659	659
fehl	704	704	646	646
rkfs	332	332	279	279
urand	223	223	225	223
solve	208	208	196	196
seval	82	82	81	81
rkf45	48	48	48	48

Figure 9.6 Number of ILOC operations for value partitioning techniques – FMM benchmark

SCC Based				
	Dominator	AVAIL	LCM	VDCM
svd	5061	5039	4444	4281
fmin	941	941	880	881
zeroin	836	836	748	748
spline	928	928	895	886
decomp	711	704	658	657
fehl	704	704	646	646
rkfs	332	332	279	279
urand	223	223	225	223
solve	186	186	187	185
seval	78	78	77	77
rkf45	35	35	35	35

Figure 9.7 Number of ILOC operations for SCC-based value numbering techniques – FMM benchmark

9.4 Normalized Instruction Counts

Table 9.3 provides a guide to Figures 9.8 through 9.21. Each of these figures compares normalized execution times for either a row or column in Table 9.1.

9.5 Comparison with Previous State of the Art

Since each of the techniques described in this thesis contributes in its own way to run-time improvements, we made a final comparison showing the sum of all contributions. Our reference point will be the combination of value partitioning and lazy code motion as presented in 1994 by Briggs and Cooper [5]. We compare this combination with SCC-based value numbering and value-driven code motion. The results are shown in Figures 9.22 and 9.23.

9.6 Compile Times

We compared the time required by each of the redundancy elimination techniques 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.

Table 9.4 compares the compile times of the three value numbering techniques. 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.

We also compared the compile-times required by each of the code removal and motion techniques. Table 9.5 compares the running times of the code removal techniques.

Type of redundancy elimination	Spec	FMM
Hash-based, vary code removal/motion	Figure 9.8	Figure 9.15
Partitioning, vary code removal/motion	Figure 9.9	Figure 9.16
SCC-based, vary code removal/motion	Figure 9.10	Figure 9.17
Dominator-based removal, vary value numbering	Figure 9.11	Figure 9.18
AVAIL-based removal, vary value numbering	Figure 9.12	Figure 9.19
Lazy code motion, vary value numbering	Figure 9.13	Figure 9.20
Value-driven code motion, vary value numbering	Figure 9.14	Figure 9.21

Table 9.3 Key to normalized instruction count figures

Figure 9.8 Comparison of hash-based value numbering techniques – Spec benchmark

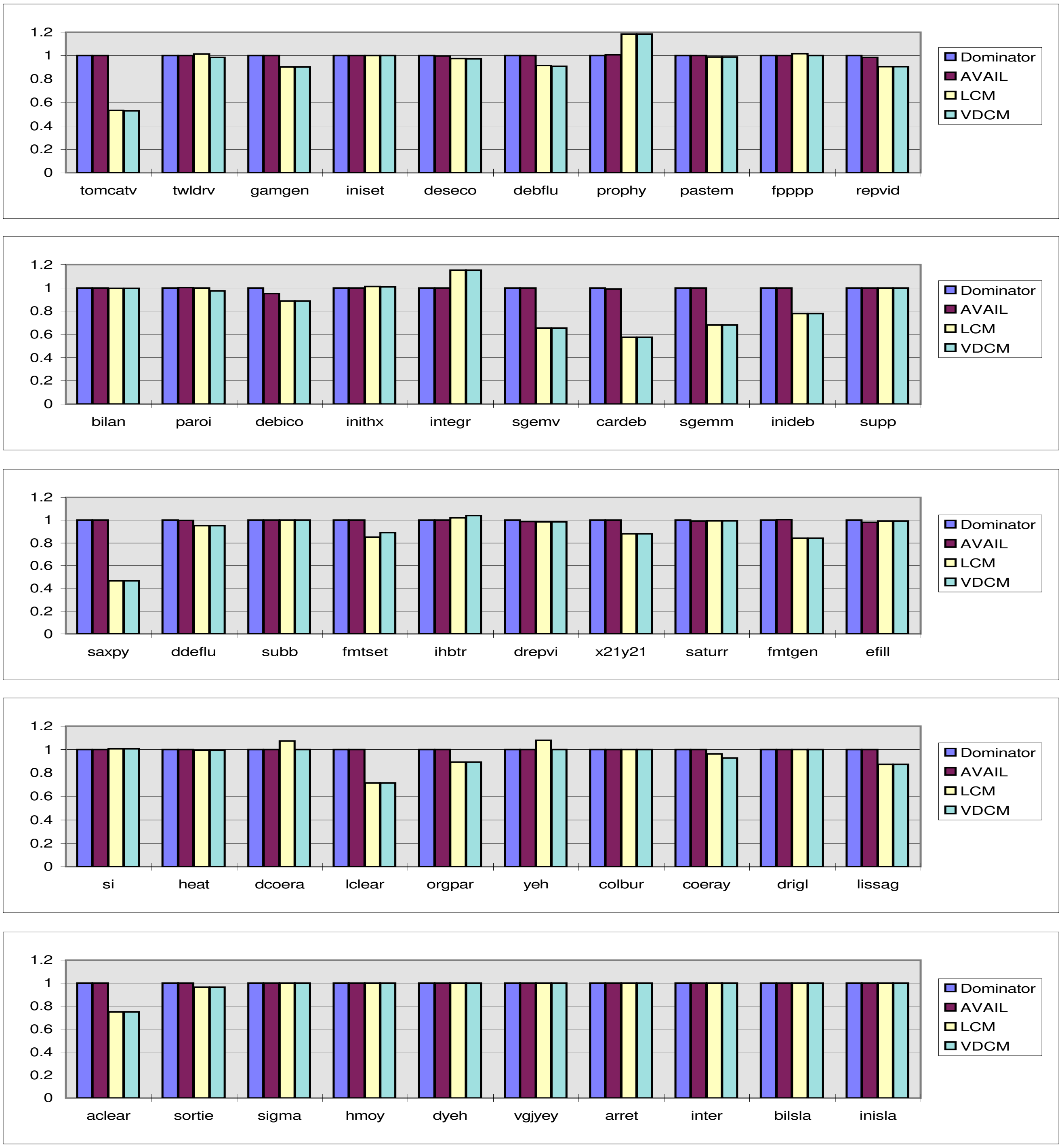


Figure 9.9 Comparison of value partitioning techniques – Spec benchmark

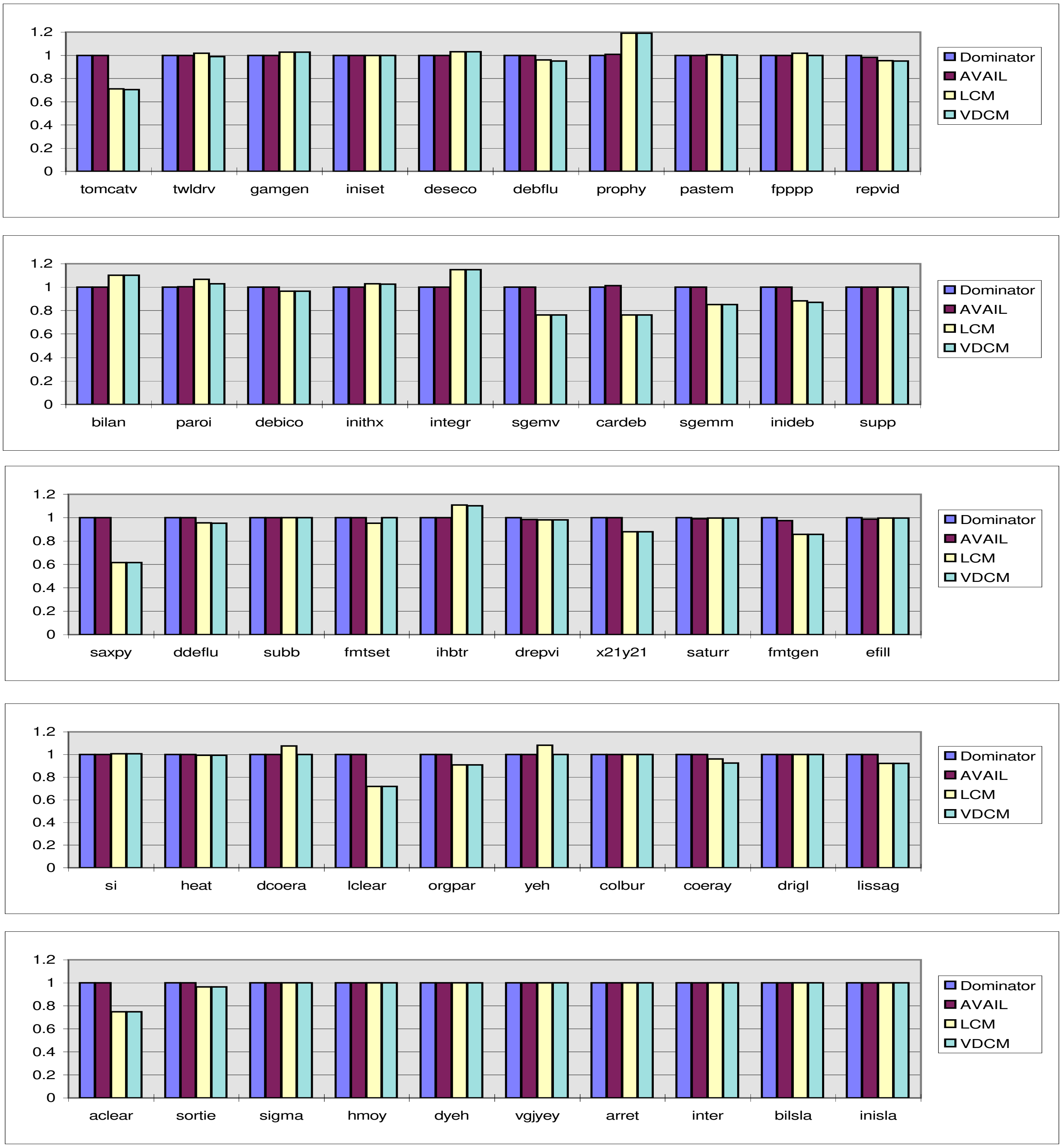


Figure 9.10 Comparison of SCC-based value numbering techniques – Spec benchmark

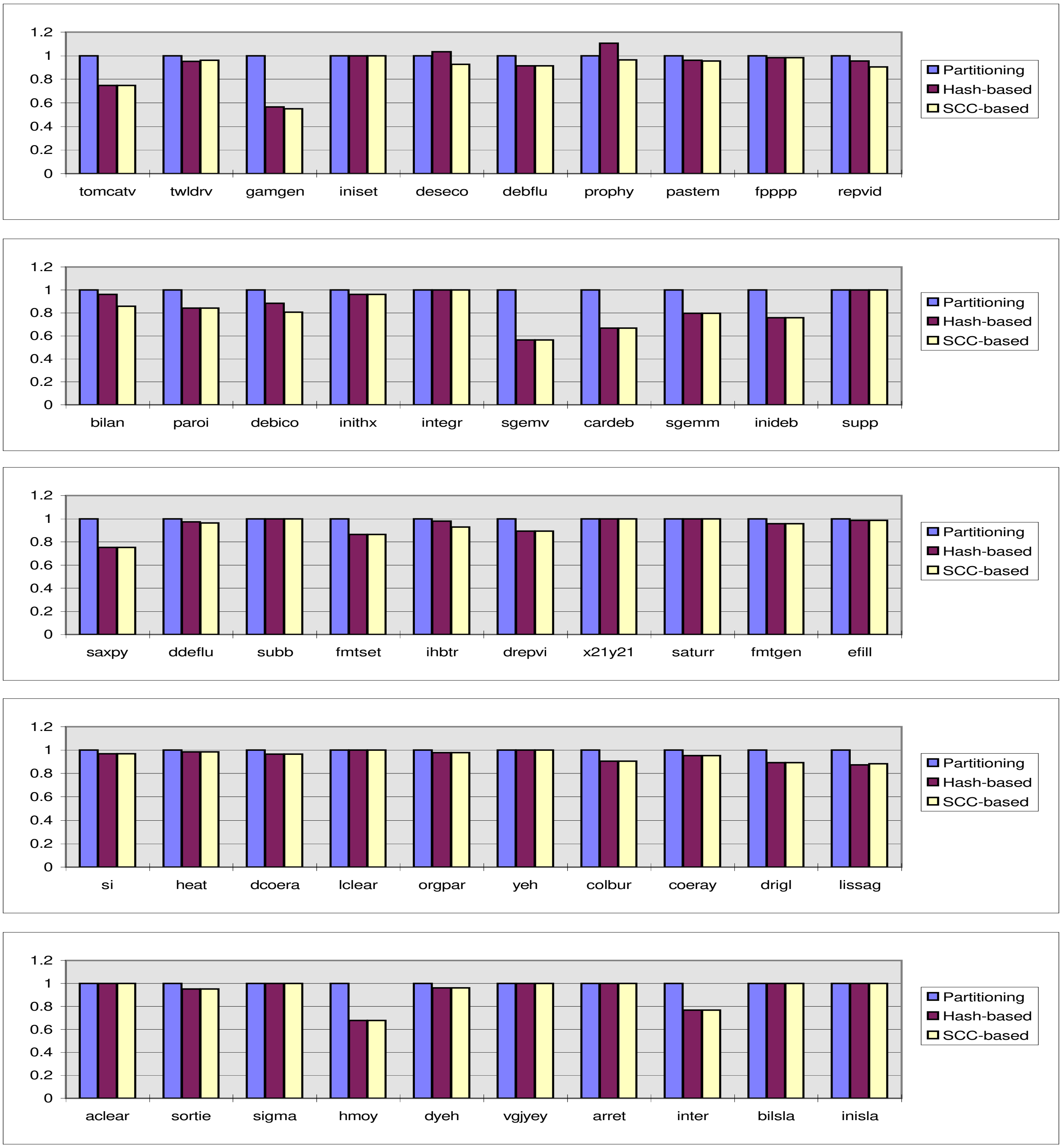


Figure 9.11 Comparison of value numbering techniques using dominator-based removal – Spec benchmark

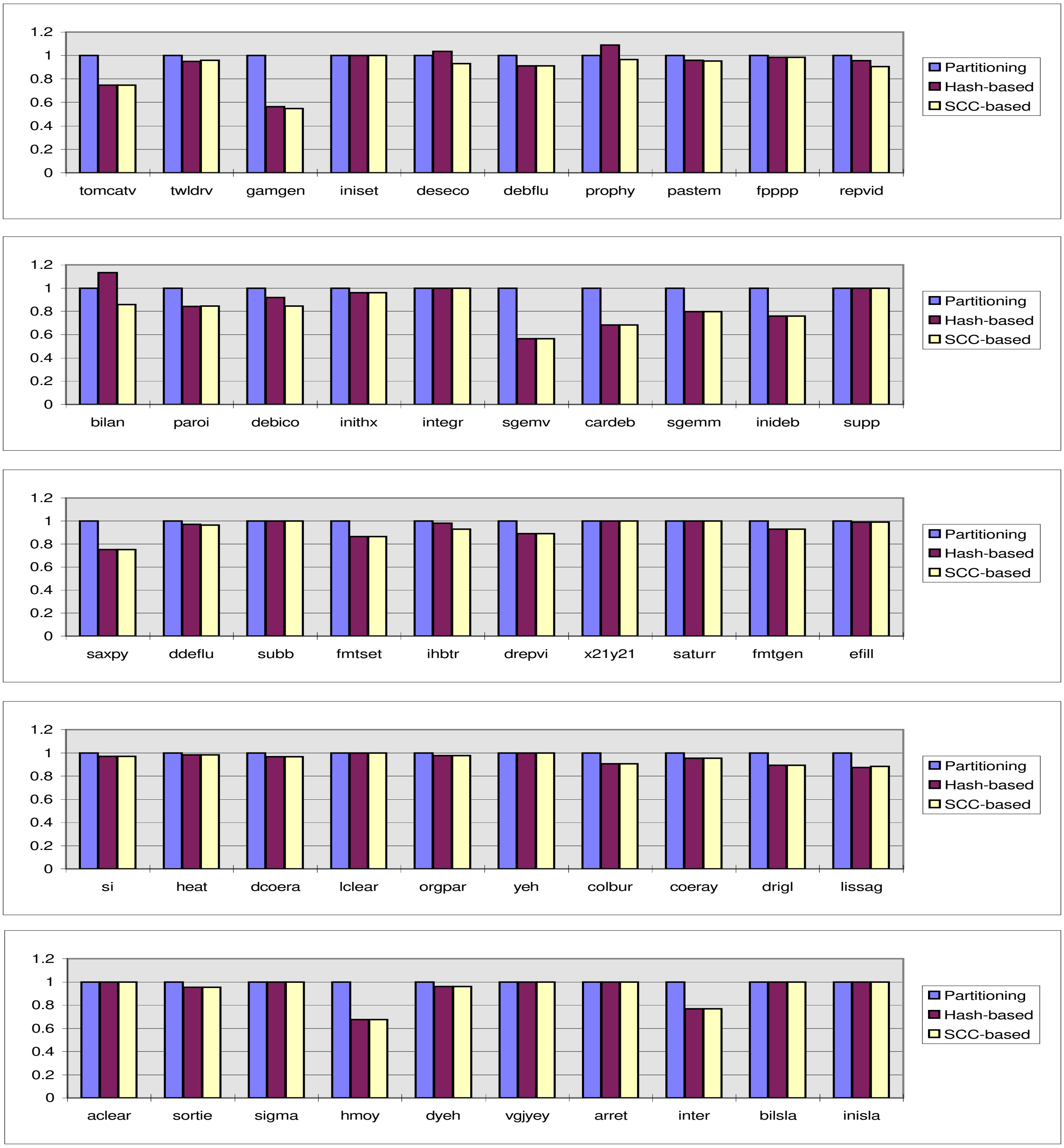


Figure 9.12 Comparison of value numbering techniques using AVAIL-based removal – Spec benchmark

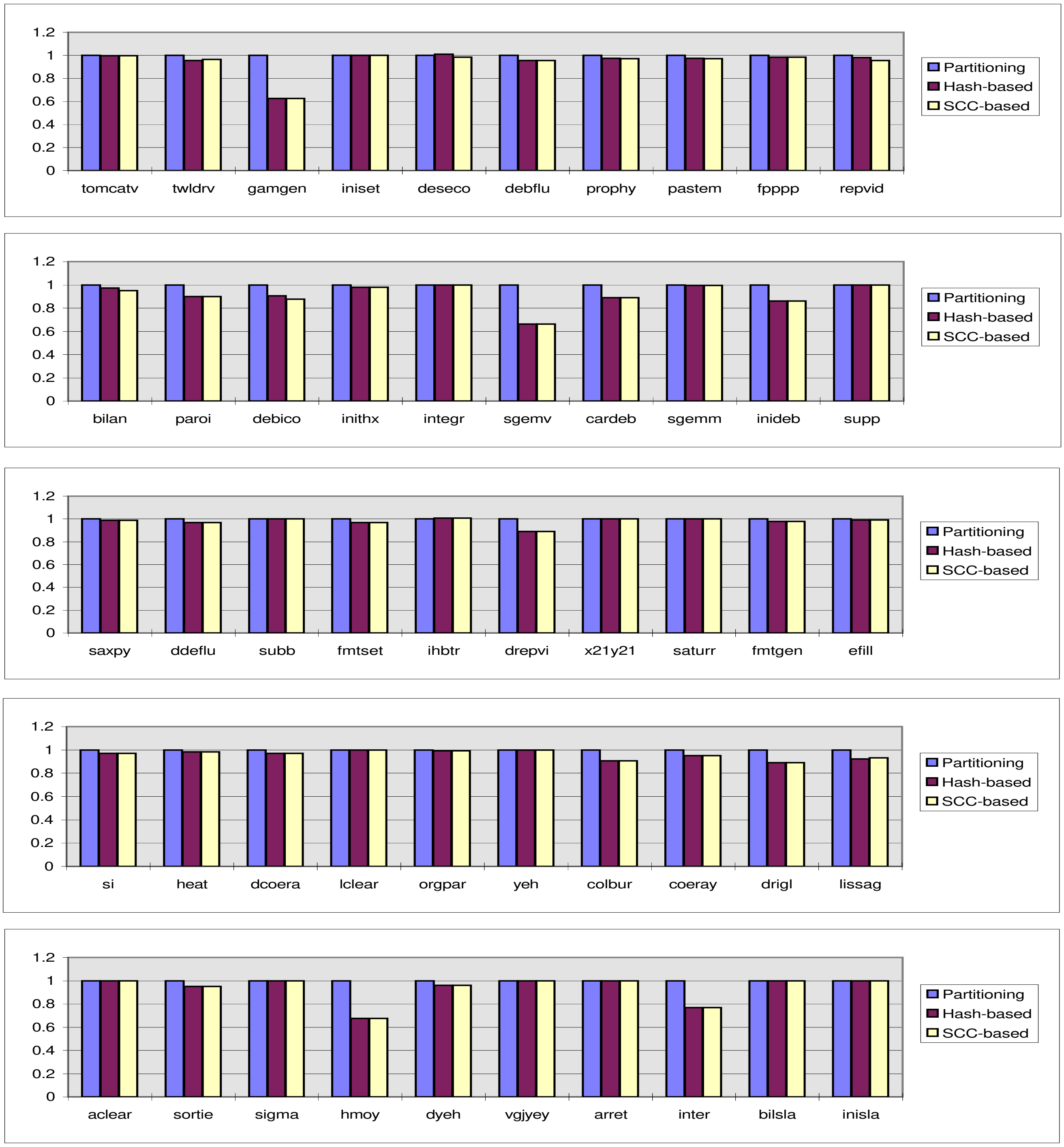


Figure 9.13 Comparison of value numbering techniques using lazy code motion – Spec benchmark

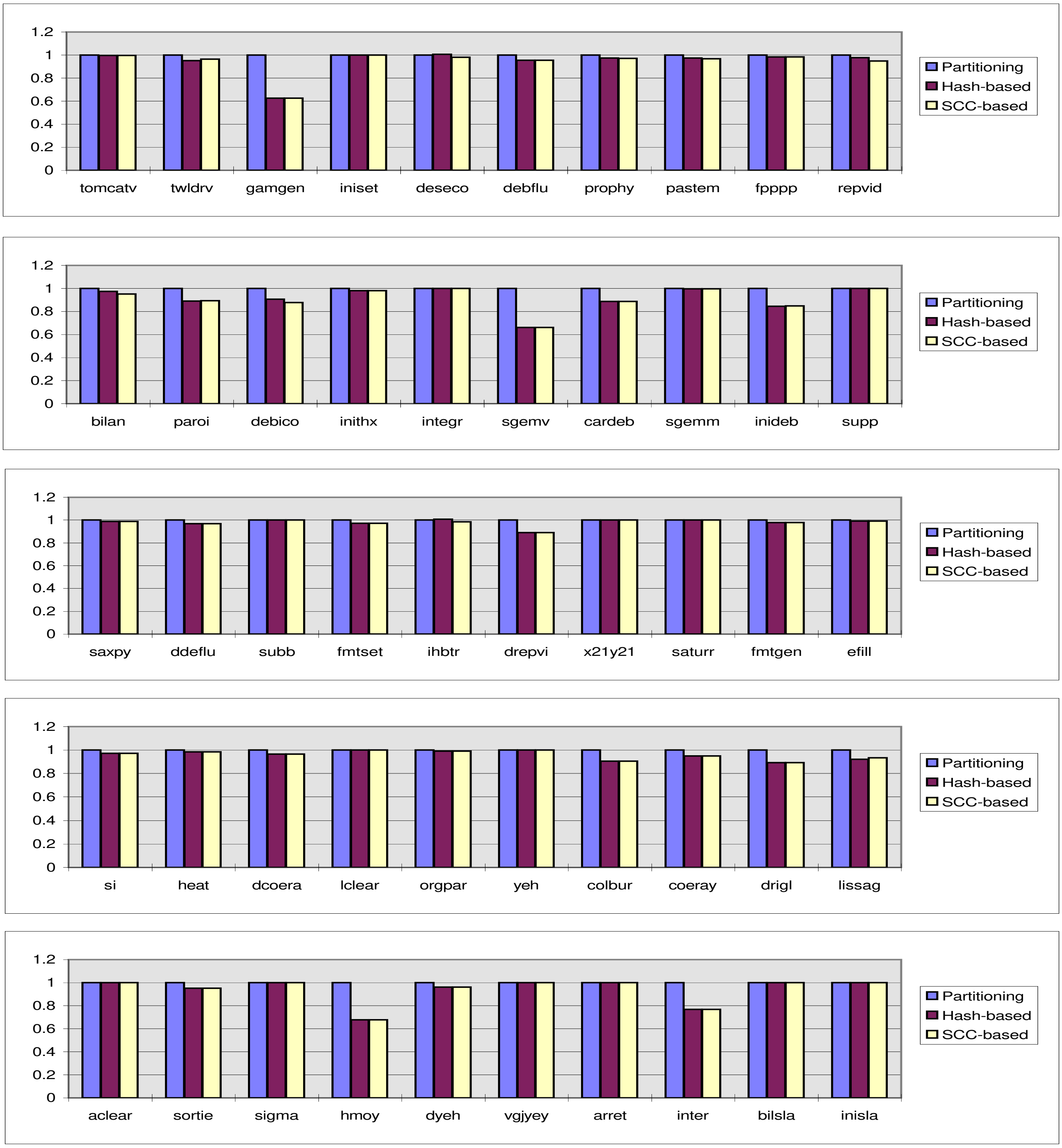


Figure 9.14 Comparison of value numbering techniques using value-driven code moion – Spec benchmark

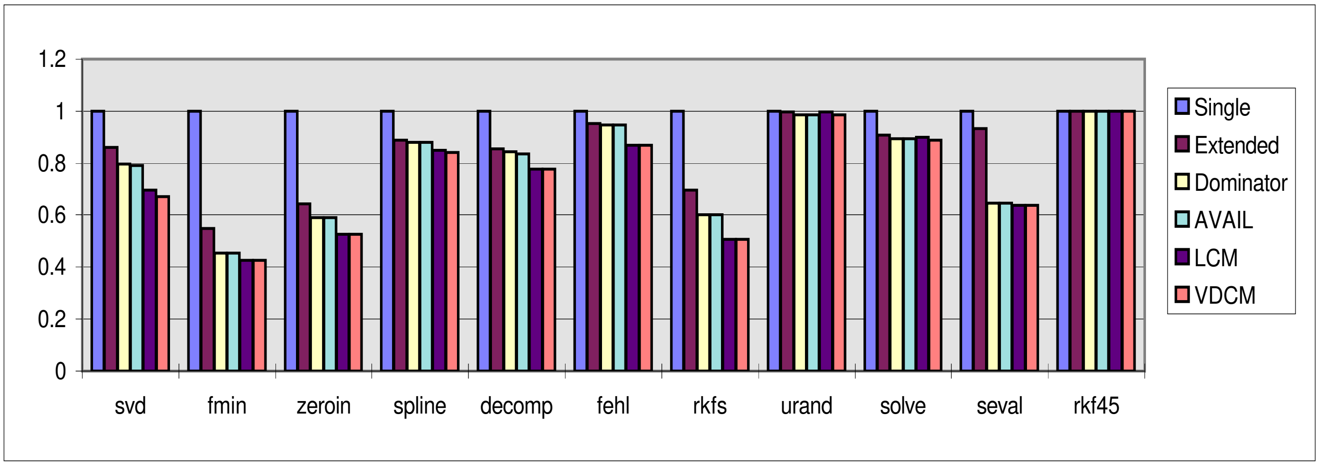


Figure 9.15 Comparison of hash-based value numbering techniques – FMM benchmark

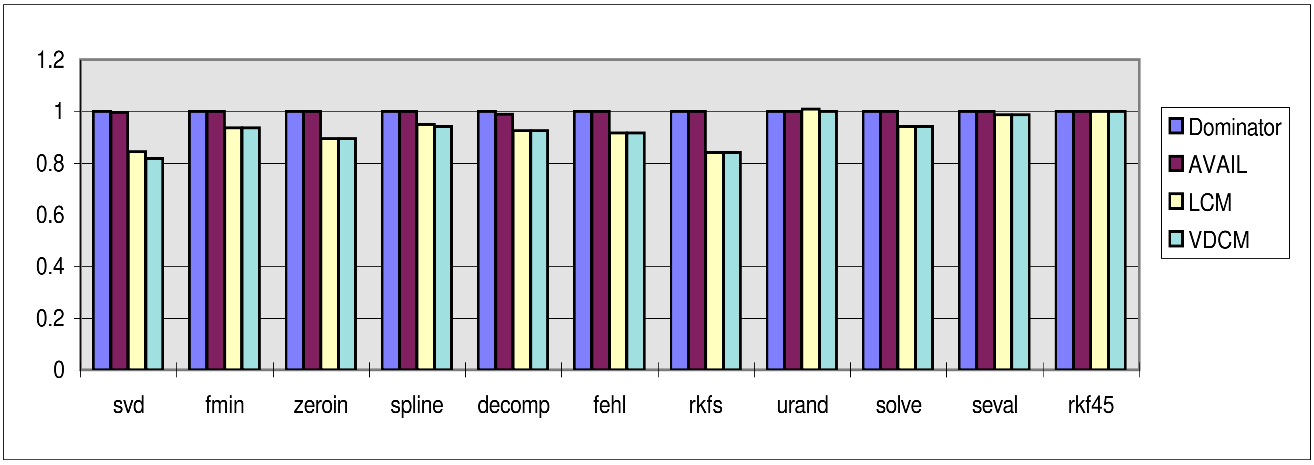


Figure 9.16 Comparison of value partitioning techniques – FMM benchmark

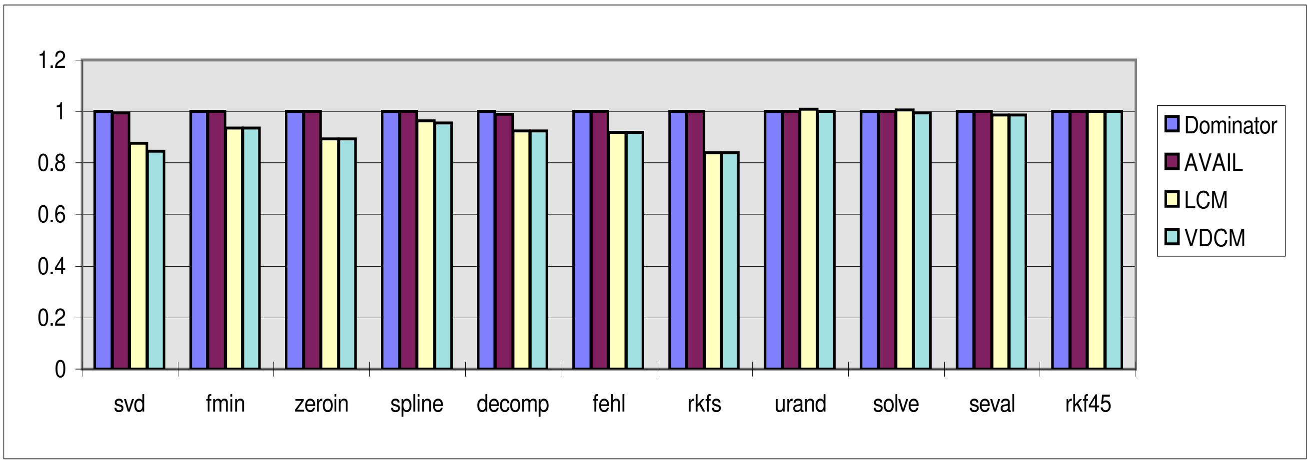


Figure 9.17 Comparison of SCC-based value numbering techniques – FMM benchmark

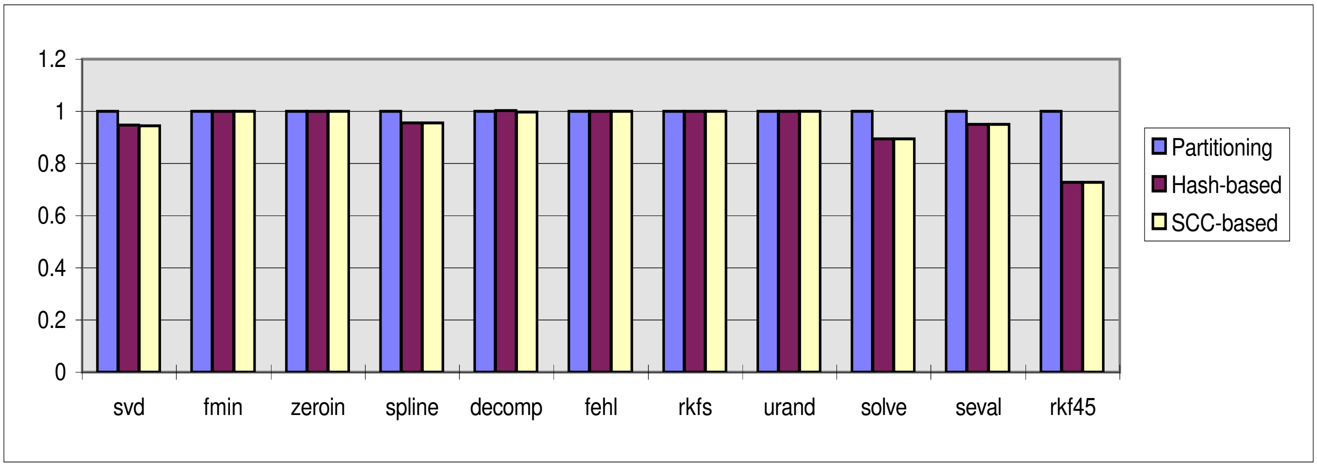


Figure 9.18 Comparison of value numbering techniques using dominator-based removal– FMM benchmark

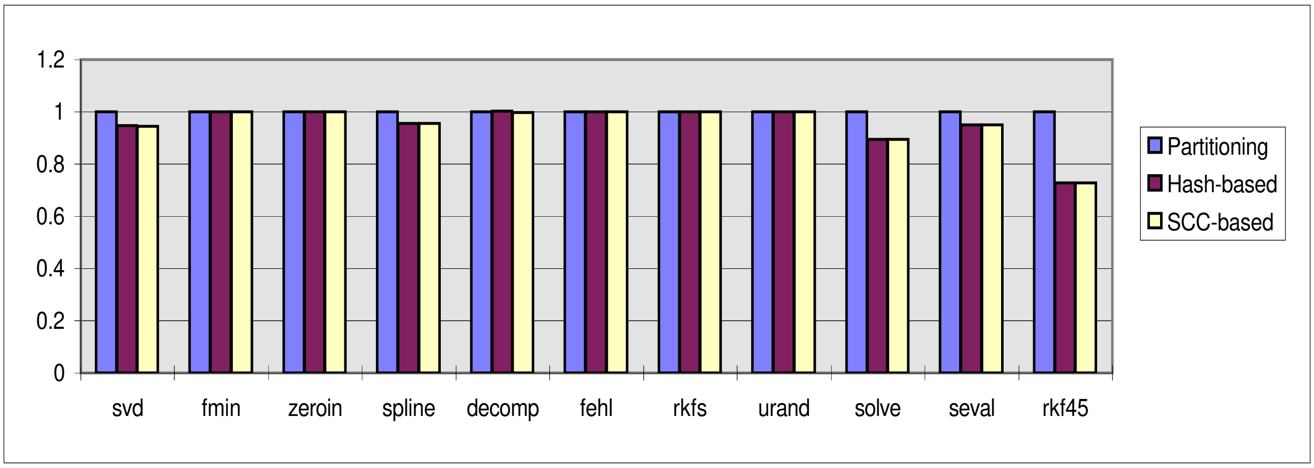


Figure 9.19 Comparison of value numbering techniques using AVAIL-based removal – FMM benchmark

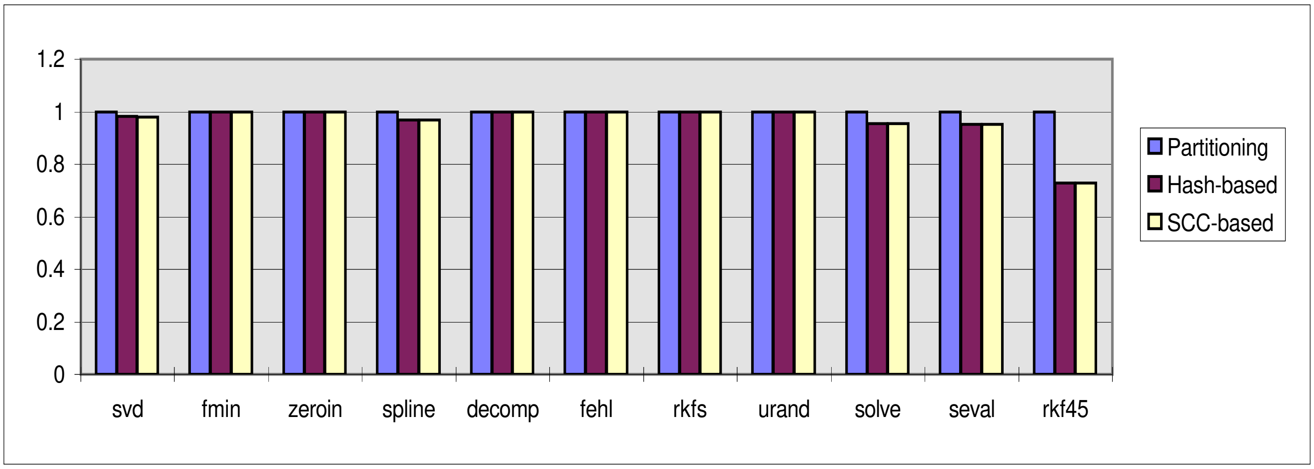


Figure 9.20 Comparison of value numbering techniques using lazy code motion – FMM benchmark

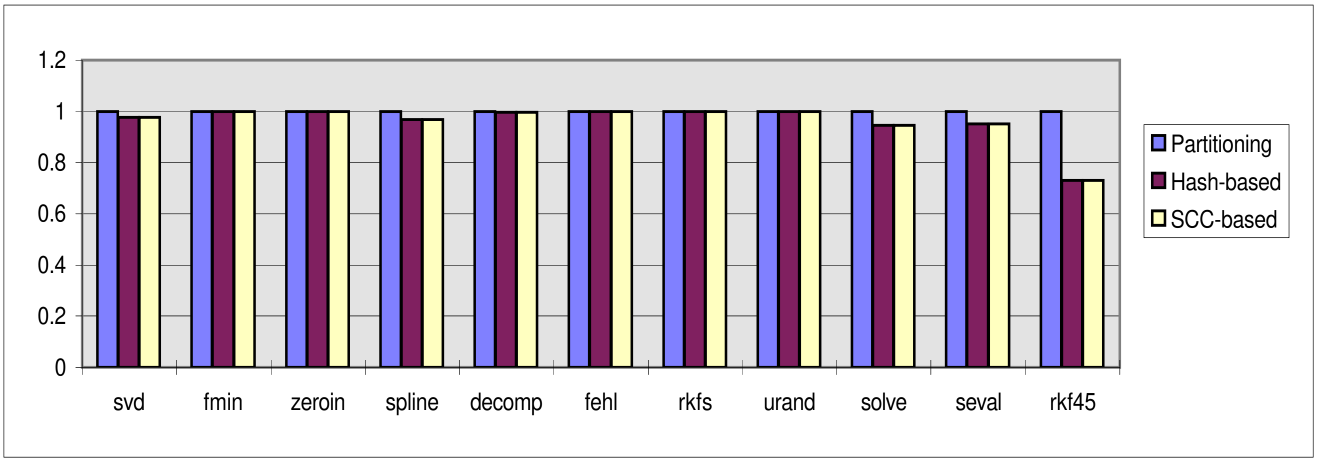


Figure 9.21 Comparison of value numbering techniques using value-driven code motion – FMM benchmark

Table 9.6 compares the running times of the code motion techniques. Since the differences in running times are determined primarily by the size of the bit vectors and the time required to compute the altered set for each block, these times are included in the comparison. Notice that the bit vector sizes used by VDCM are larger than those used by LCM. This is due to the fact that there are more values than lexical names (*i.e.*, the same name can have many values). However, the reduction in the time required to compute the altered sets more than compensates for this difference. In every instance, VDCM ran faster than LCM.

routine	*blocks*	*SSA names*	*operations*	*hash-based*	*SCC-based*	*partitioning*
`tomcatv`	131	3366	3326	0.05	0.08	0.07
`ddeflu`	109	8994	6873	0.11	0.46	0.81
`debflu`	116	7147	4514	0.08	0.48	0.93
`deseco`	251	16502	13121	0.30	0.81	1.85
`twldrv`	261	26913	15168	0.40	3.11	6.09
`fpppp`	2	26588	25934	0.63	0.65	1.16

Table 9.4 Compile times of value numbering techniques

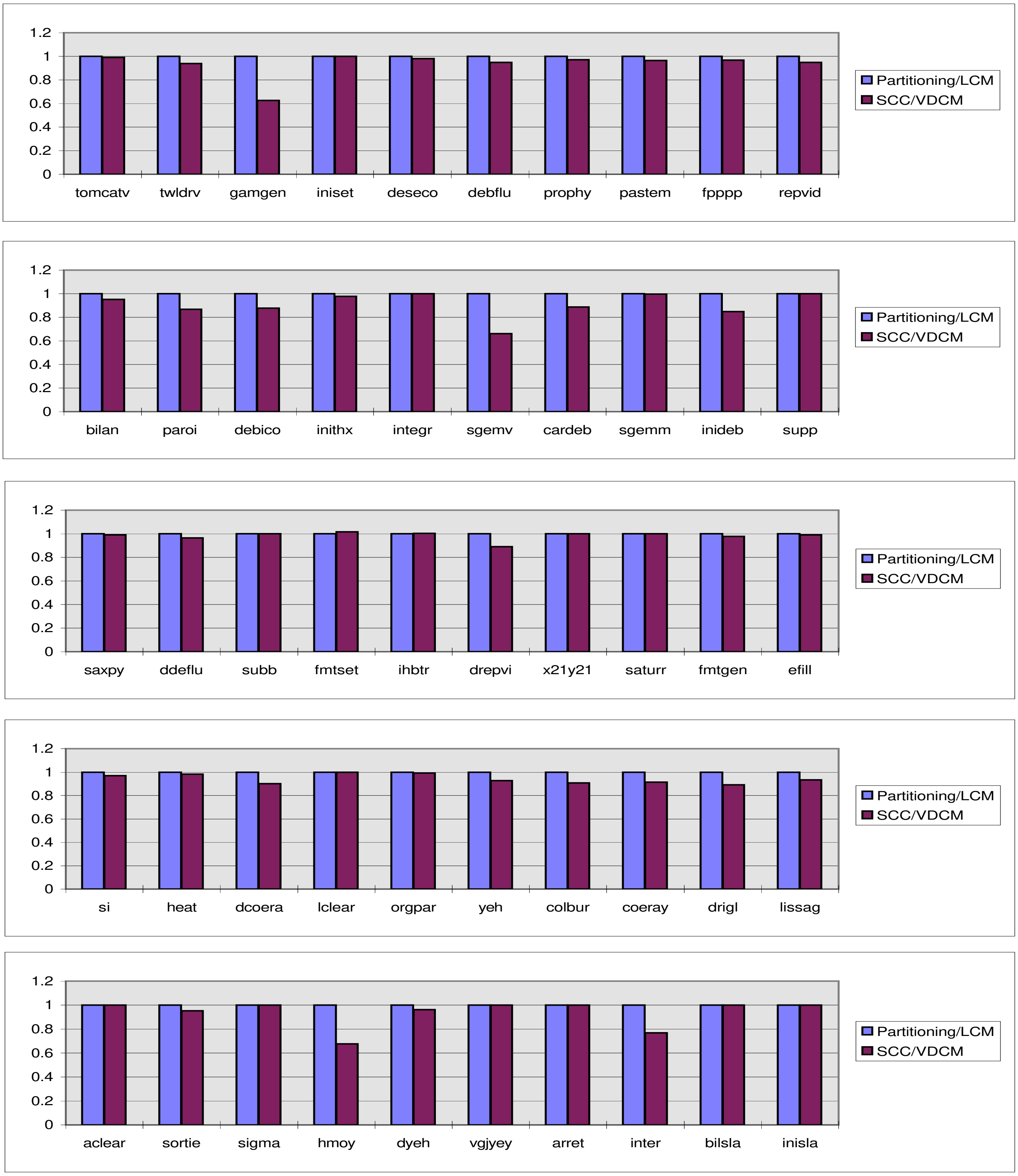


Figure 9.22 Comparison with previous "state of the art" – Spec benchmark

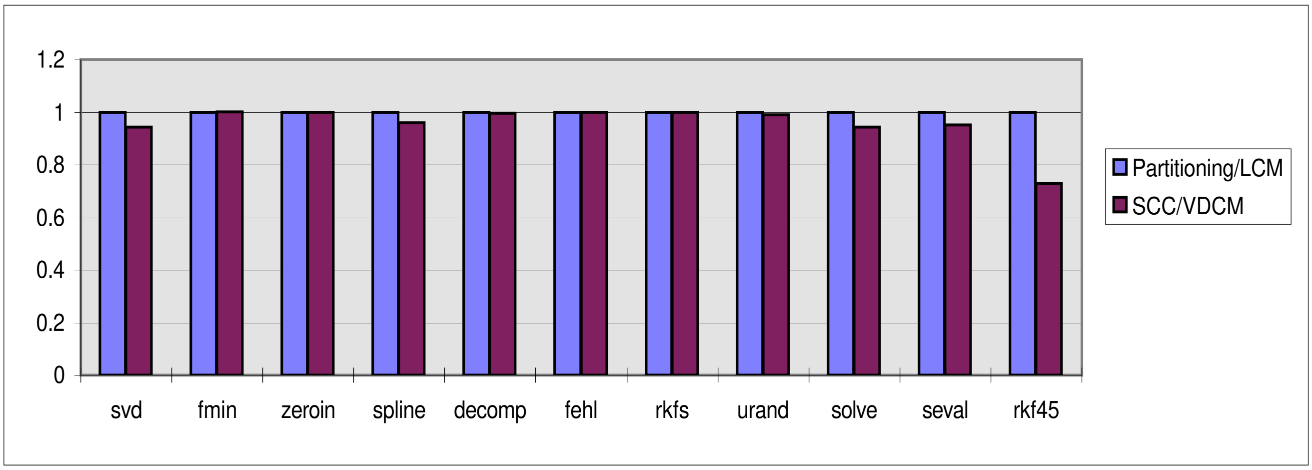


Figure 9.23 Comparison with previous "state of the art" – FMM benchmark

routine	*blocks*	*SSA names*	*operations*	*dominator*	*AVAIL*
`tomcatv`	131	3366	3326	0.01	0.01
`ddeflu`	109	8994	6873	0.03	0.03
`debflu`	116	7147	4514	0.01	0.03
`deseco`	251	16502	13121	0.08	0.09
`twldrv`	261	26913	15168	0.09	0.21

Table 9.5 Compile times of code removal techniques

	LCM			VDCM		
routine	*set size*	*altered*	*total*	*set size*	*altered*	*total*
`tomcatv`	790	0.39	0.50	924	0.06	0.19
`ddeflu`	1436	1.33	1.47	3648	0.11	0.49
`debflu`	1101	0.76	0.88	3879	0.11	0.49
`deseco`	2513	4.20	4.76	6067	0.51	1.85
`twldrv`	3928	9.26	10.17	16140	0.93	4.77

Table 9.6 Compile times of code motion techniques

9.7 Relief of Register Pressure

Figures 9.24 through 9.27 compare each of the heuristics for relief of register pressure (see Chapter 8) to allocation without relief for the routines in our test suite. Since our goal is to replace the memory operations that would be inserted during register allocation with less expensive computations, we perform comparisons with LOAD and STORE operations weighted by 1 and then by 3. As the weight is increased, further improvements are seen until they eventually level off. The code generated in this experiment was instrumented not only to count the total number of instructions executed but also to count the number of LOAD and STORE instructions executed. We then use this secondary count to vary the weight of memory operations. In our experiments, heuristics 2 and 4 perform consistently well.

9.8 Summary

This chapter presents experimental data comparing the effectiveness of the techniques presented in this thesis. Experiments are run in the context of our optimizing compiler, and the number of ILOC operations executed are measured. The basis for each experiment is varying only a single pass in the sequence of optimizations.

The first set of experiments compares the effectiveness of various redundancy elimination techniques. In general, each refinement to the technique results in an improvement in the results. When comparing techniques for value numbering, we see that hash-based value numbering almost always eliminates significantly more redundancies than value partitioning. SCC-based value numbering removes slightly more redundancies than hash-based value numbering. When comparing the techniques for code removal and motion, 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. AVAIL-based removal is slightly better than dominator-based removal. The differences between AVAIL-based removal and lazy code motion are significant. This is because moving invariant code out of loops provides a great deal of improvement. However, there are some examples where AVAIL-based removal is better than lazy code motion because it is based on values while lazy code motion is based on lexical names. The differences between lazy code motion and value-driven code motion are small because lazy code motion is a provably optimal technique.

Figure 9.24 Comparison of relief heuristics – Spec benchmark, LOAD/STORE weight = 1

Figure 9.25 Comparison of relief heuristics – Spec benchmark, LOAD/STORE weight = 3

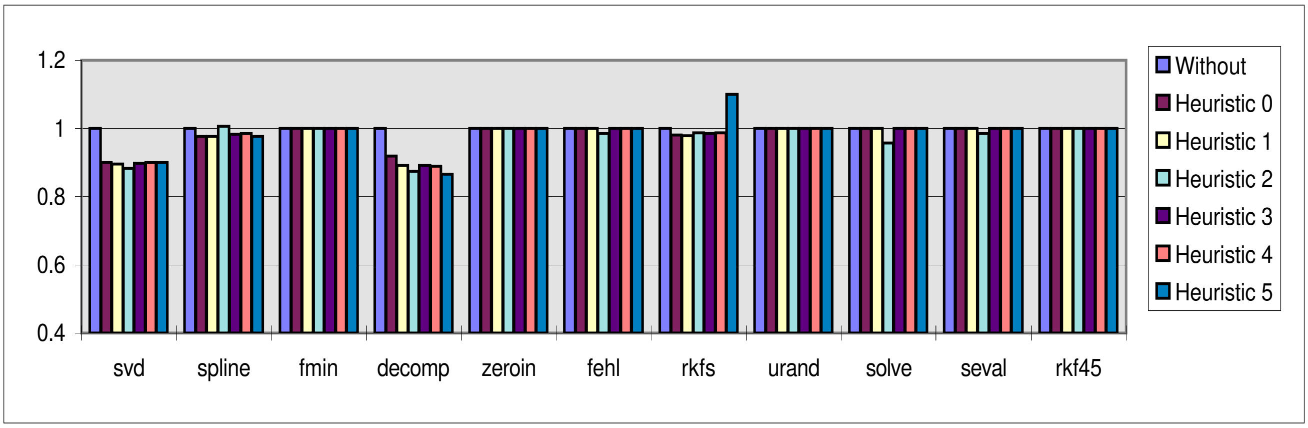


Figure 9.26 Comparison of relief heuristics – FMM benchmark, LOAD/STORE weight = 1

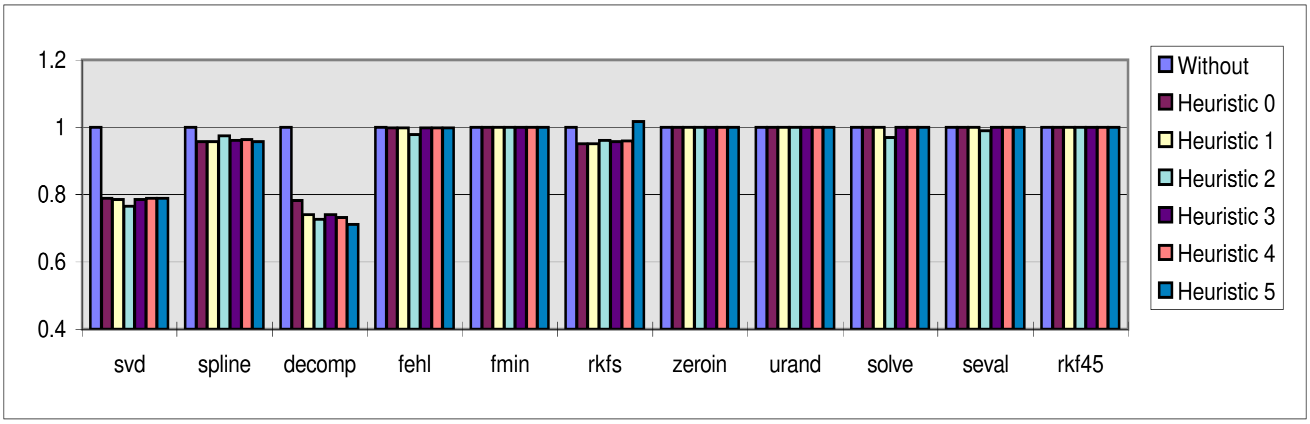


Figure 9.27 Comparison of relief heuristics – FMM benchmark, LOAD/STORE weight = 3

Anomalous behavior is seen in a few instances. There are three reasons to account for these results:

1. Removing more redundancies can result in partially dead code (see Section 2.5).
2. Changes in the live ranges can inhibit copy coalescing (see Chapter 8).
3. Improved redundancy elimination introduces more opportunities for operator strength reduction (see Appendix A).

Negative results from improved redundancy elimination are never more than a few percent while positive results can be as much as 40%.

The second set of experiments compares the heuristics for relief of register pressure. The amount of improvement depends on the cost of LOAD and STORE instructions relative to register-to-register instructions. The higher the cost, the greater the improvement. When the register pressure is high, each of the heuristics will improve the performance except in a few anomalous cases. Improvements as high as 60% are seen when the cost of memory operations is three times the cost of register-to-register operations. However, decreased in performance are also seen in a few cases.

9.9 Recommendations

When choosing algorithms to implement in a compiler, the developer must weigh the tradeoffs between three areas:

1. Execution time of the optimized program
2. Execution time of the compiler
3. Amount of programmer time required for implementation

For each of these areas, this thesis has provided the information necessary for the compiler writer to make an informed decision about which form of redundancy elimination to implement. The decision should be based engineering decisions and the amount of importance placed on each of the above areas.

For the value numbering phase, the choice is between hash-based value numbering, value partitioning, and SCC-based value numbering. Value partitioning can be eliminated from consideration because it is more difficult to implement; it runs slower, and it eliminates fewer redundancies in practice than the other two options. The choice between hash-based and SCC-based value numbering is not so clear. Even though SCC-based value numbering will find more redundancies, our experiments indicate that the improvements made in practice are small. On the other hand, hash-based value numbering can run significantly faster than SCC-based value numbering. We can characterize the types of redundancies identified by SCC-based value numbering that are not discovered by hash-based value numbering as equal values involved in a cycle (*e.g.*, two induction variables with the same initial value that are incremented by the same amount each trip through a loop). We do not expect to see these types of redundancies in code written by programmers. In fact, in our experiments, no code generated by the front-end contains these types of redundancies. However, certain transformations applied by the compiler can introduce them. In our experiments, the operator strength reduction pass can introduce them even though it is fairly careful not to introduce redundancies. Other optimizations such as loop unrolling and transformations intended to improve cache performance can introduce them [10]. If any of these transformations are performed before redundancy elimination, then the compiler writer might give more weight to SCC-based value numbering.

For the code motion phase, the choice is between lazy code motion and value-driven code motion. Value-driven code motion is the clear winner because it wins in all three areas. On the other hand, if an implementation of lazy code motion already exists and programmer time is at a premium, the compiler writer might consider using the existing implementation and accepting the loss in optimization time and performance.

Selecting which heuristic is appropriate for relief of register pressure will depend on the characteristics of the target machine as well as the register allocator. In our experiments, heuristics 2 and 4 perform consistently well. On large routines, heuristic 2 performs better than the other heuristics. However, this heuristic can increase the running time of small routines. This is due to the fact that it can move code into an empty block that could have otherwise been removed (see Figure 8.4). The result is an increase in the number of JUMP instructions executed. If the target processor has an instruction prefetch buffer with zero-cost JUMPs, then this effect will not matter. In this case, heuristic 2 would be the clear choice. On the other hand, heuristic 4 is simple to implement, and it did not increase the running time of any routine in our test suite. There may be some combination of compiler and processor where this heuristic performs best.

Chapter 10

Summary of Contributions

Optimizing compilers apply a sequence of transformations to a routine in order to improve the performance of the target program. One or more of these passes is devoted to redundancy elimination. The primary focus of this thesis has been to improve upon known techniques for redundancy elimination. Redundancies can be eliminated by removing instructions or moving them to less frequently executed locations. In the past, the algorithms for removing computations have been designed and implemented independently from algorithms for moving instructions. The primary contribution of this thesis is to unify these techniques. By understanding how these two optimizations interact, we can simplify each of them, and the resulting combination will be more powerful than the sum of the two parts.

Value numbering attempts to discover which instructions compute the same value. It assigns numbers to values in such a way that two values are assigned the same number if the compiler can prove they are equal. There are three main approaches to value numbering:

Hash-based value numbering This algorithm uses a hash table to map expressions to value numbers. It was originally applied to single basic blocks and extended basic blocks. We have extended this technique to operate over the dominator tree and to use a unified hash table.

Value partitioning This algorithm uses a variation of Hopcroft's DFA minimization algorithm to partition the values in a routine into congruence classes. We have extended this technique to handle commutative operations and to eliminate redundant stores.

SCC-based value numbering This is an original technique that combines the features of hash-based value numbering and value partitioning. It is easy to understand and to implement, it can handle constant folding and algebraic identities, and it is global. We can prove that the algorithm finds at least as many congruences as hash-based value numbering or value partitioning.

Value numbering will renumber the registers and ϕ-nodes in a routine so that congruent values are given the same number. However, renumbering alone will not improve the running time of the routine; we must remove computations or move them to less frequently executed locations.

Dominator-based removal This technique was suggested by Alpern, Wegman, and Zadeck. It removes instructions that are dominated by a congruent computation.

AVAIL-based removal This technique relies on data-flow analysis to discover the set of available expressions and remove instructions whose value is in the set. We have improved this technique so that it operates on values rather than lexical names.

Partial redundancy elimination/Lazy code motion These algorithms combine loop invariant code motion with common subexpression elimination. They rely on data-flow analysis to move instructions. We have extended these techniques to allow motion of LOAD instructions.

Value-driven code motion This original approach to data-flow analysis is based on values rather than lexical names. Traditional data-flow frameworks cannot move an instruction past a definition of one of its subexpressions. This restriction can be relaxed when value numbering has identified the computations that produce redundant values. By understanding how code motion interacts with value numbering, we can simplify and improve the code motion framework. This algorithm is faster than lazy code motion and it can eliminate more redundancies.

Redundancy elimination can change the live ranges in a routine which can potentially have a negative impact on register allocation. We have invented a new technique to relieve register pressure by reintroducing redundancies into the routine. As a result, less spill code will be inserted during register allocation, and the running time of the routine is reduced.

A theoretical comparison of the redundancy elimination techniques reveals that SCC-based value numbering and value-driven code motion are never worse than their competitors in terms of eliminating redundancies. However, an equally important question is how much this theoretical distinction matters in practice. Chapter 9 experimentally compares all of the techniques described in this thesis.

Appendix A gives a new algorithm for operator strength reduction that improves on existing algorithms in several ways. It takes advantage of the properties of SSA form and the "shape" of a routine after redundancy elimination.

Appendix A

Operator Strength Reduction

Operator strength reduction is a transformation that a compiler uses to replace costly (strong) instructions with cheaper (weaker) ones. The algorithm replaces an iterated series of strong computations with an equivalent series of weaker computations. The classic example replaces certain multiplication operations inside a loop with equivalent addition operations. This case arises routinely in loop-based array address calculations, and many other operations can be reduced in this manner. Allen, Cocke, and Kennedy provide a detailed catalog of such reductions [3].

Strength reduction has been an important transformation for two principal reasons. First, multiplying integers has usually taken longer than adding them. This made strength reduction profitable; the amount of improvement varied with the relative costs of addition and multiplication. Second, strength reduction decreased the "overhead" introduced by translation from a high-level language down to assembly code. Opportunities for this transformation are frequently introduced by the compiler as part of address translation for array elements. In part, strength reduction's popularity stems from the fact that these computations are plentiful, stylized, and, in a very real sense, outside the programmer's concern.

In the future, we may see microprocessors where an integer multiply and an integer add both take a single cycle. On such a machine, strength reduction will still have a role to play. In combination with algebraic reassociation [16, 40, 5], strength reduction may let the compiler use fewer induction variables in a loop, lowering both the operation count inside the loop and the demand for registers. This effect may be especially pronounced in code that has been automatically blocked to improve locality [46, 9].

This chapter presents a new algorithm for performing strength reduction. It produces results similar to those of Allen, Cocke, and Kennedy's classic algorithm [3]. By assuming some specific prior optimizations and operating on the SSA form of the procedure [18], we have derived a method that (1) is simple to understand and to implement, (2) relies on the dominator tree which must be computed during SSA

```
sum = 0.0
do i = 1, 100
        sum = sum + a(i)
enddo
```

Source code

$$
\begin{array}{rl}
 & sum \leftarrow 0.0 \\
 & i \leftarrow 1 \\
L\!: & t1 \leftarrow i - 1 \\
 & t2 \leftarrow t1 \times 4 \\
 & t3 \leftarrow t2 + a \\
 & t4 \leftarrow \texttt{load}\ t3 \\
 & sum \leftarrow sum + t4 \\
 & i \leftarrow i + 1 \\
 & \textbf{if}\ (i \leq 100)\ \textbf{goto}\ L
\end{array}
$$

Intermediate code

$$
\begin{array}{rl}
 & sum_0 \leftarrow 0.0 \\
 & i_0 \leftarrow 1 \\
L\!: & sum_1 \leftarrow \phi(sum_0, sum_2) \\
 & i_1 \leftarrow \phi(i_0, i_2) \\
 & t1_0 \leftarrow i_1 - 1 \\
 & t2_0 \leftarrow t1_0 \times 4 \\
 & t3_0 \leftarrow t2_0 + a \\
 & t4_0 \leftarrow \texttt{load}\ t3_0 \\
 & sum_2 \leftarrow sum_1 + t4_0 \\
 & i_2 \leftarrow i_1 + 1 \\
 & \textbf{if}\ (i_2 \leq 100)\ \textbf{goto}\ L
\end{array}
$$

SSA form

Figure A.1 Example

construction rather than the loop structure of the program (which can be costly to compute), (3) avoids instantiating the sets of induction variables and region constants required by other algorithms, (4) processes each candidate instruction immediately rather than maintaining a worklist, and (5) greatly simplifies linear function test replacement. Its asymptotic complexity is, in the worst case, the same as the Allen, Cocke, and Kennedy algorithm.

Opportunities for strength reduction arise routinely from details that the compiler inserts as it converts from a source-level representation to a machine-level representation. To see this, consider the simple Fortran code fragment shown in Figure A.1. The left column shows source code; the central column shows a low-level intermediate code version of the same loop. Notice the four instruction sequence that begins at the label L. The compiler inserted this code (with its multiply) as the expansion of `a(i)`. The right column shows the code in pruned SSA form.

The left column of Figure A.2 shows the result of strength reduction and the optimizations described in Section A.2.1. The compiler created a new variable $t5$ to hold the value of the expression $(i-1)4+a$. We will describe an algorithm to automate this process inside a compiler.

Of course, further improvement may be possible. For example, if the only remaining use for i_2 is in the control-flow tests that govern the loop, the compiler could reformulate the tests to use $t5_2$, making the instructions that define i useless (or

$$
\begin{array}{ll}
 & sum_0 \leftarrow 0.0 \\
 & i_0 \leftarrow 1 \\
 & t5_0 \leftarrow a \\
L: & sum_1 \leftarrow \phi(sum_0, sum_2) \\
 & i_1 \leftarrow \phi(i_0, i_2) \\
 & t5_1 \leftarrow \phi(t5_0, t5_2) \\
 & t4_0 \leftarrow \texttt{load}\, t5_1 \\
 & sum_2 \leftarrow sum_1 + t4_0 \\
 & i_2 \leftarrow i_1 + 1 \\
 & t5_2 \leftarrow t5_1 + 4 \\
 & \textbf{if}\ (i_2 \leq 100)\ \textbf{goto}\ L
\end{array}
$$

After strength reduction

$$
\begin{array}{ll}
 & sum_0 \leftarrow 0.0 \\
 & t5_0 \leftarrow a \\
L: & sum_1 \leftarrow \phi(sum_0, sum_2) \\
 & t5_1 \leftarrow \phi(t5_0, t5_2) \\
 & t4_0 \leftarrow \texttt{load}\, t5_1 \\
 & sum_2 \leftarrow sum_1 + t4_0 \\
 & t5_2 \leftarrow t5_1 + 4 \\
 & \textbf{if}\ (t5_2 \leq 396 + a)\ \textbf{goto}\ L
\end{array}
$$

After linear function test replacement

Figure A.2 Transformed code

"dead"). This transformation is called *linear function test replacement* (LFTR). The results of applying LFTR appear in the right column of Figure A.2.

A.1 The Algorithm

A.1.1 Preliminary Transformations

To simplify the algorithm, we assume that some prior optimization has been performed. This preprocessing simplifies the task of strength reduction by encoding certain facts in the "shape" of the code. For example, after invariant code has been moved out of loops, we can easily identify loop invariant values based on the location of their definition.

Chapters 2 through 7 describe the algorithms for redundancy elimination which combines loop invariant code motion and common subexpression elimination. We perform global reassociation and global renaming prior to redundancy elimination [5].

Because our algorithms for redundancy elimination will not move conditionally executed code out of loops, it is possible that some loop invariant code will remain inside a loop. We account for this by defining a region constant to be *either* a compile-time constant or a value whose definition is outside the loop. This does not identify all possible region constants, but our experiments indicate that the missed opportunities are insignificant in practice. Since we consider compile-time constants to be region constants, we require some form of *constant propagation* to identify as many constants as possible. We use Wegman and Zadeck's sparse conditional constant algorithm [44].

We construct the *pruned* SSA *form* of the program [18]. In the program's SSA graph, each node represents an operation or a ϕ-node, and edges flow from uses to definitions. The SSA graph can be built from the resulting program by adding the use-definition chains, which can be represented as a lookup table indexed by SSA names. Figure A.3 shows the SSA graph for the example program in Figure A.1.

A.1.2 Finding Region Constants and Induction Variables

Previous strength reduction algorithms have been centered around loops, or *regions*, inside a procedure. These are detected using Tarjan's flow-graph reducibility algorithm [43]. Given a region r, a node in the SSA graph is a *region constant* with respect to r if its value does not change inside r. A variable is an *induction variable* with respect to region r if within the region r its value is only incremented or decremented by a region constant. We avoid the need to build the loop tree by using the dominator tree that was constructed during the conversion to SSA form [33]. We take advantage of two key properties of SSA form to identify region constants and induction variables.

1. After invariant code has been moved out of loops, each region constant with respect to region r will either be a compile-time constant or its definition will strictly dominate every block in r. The SSA construction algorithm ensures this; if it is not true, the SSA form must have a ϕ-node inside r.

2. Every induction variable forms a strongly connected component (SCC) in the SSA graph. Notice that the converse is not true (*i.e.*, not every SCC represents an induction variable). In Figure A.3, the SCC containing sum_1 and sum_2 does not represent an induction variable because $t4_0$ is not a region constant.

The idea of finding induction variables as SCCs of the SSA graph is due to Wolfe [47]. To discover the SCCs, we will use Tarjan's algorithm based on depth-first search, shown in Figure A.4 [42]. It 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 one or more nodes is popped from the stack, all of the operands referenced in those nodes that are defined outside the collection have already been popped. We can capitalize on

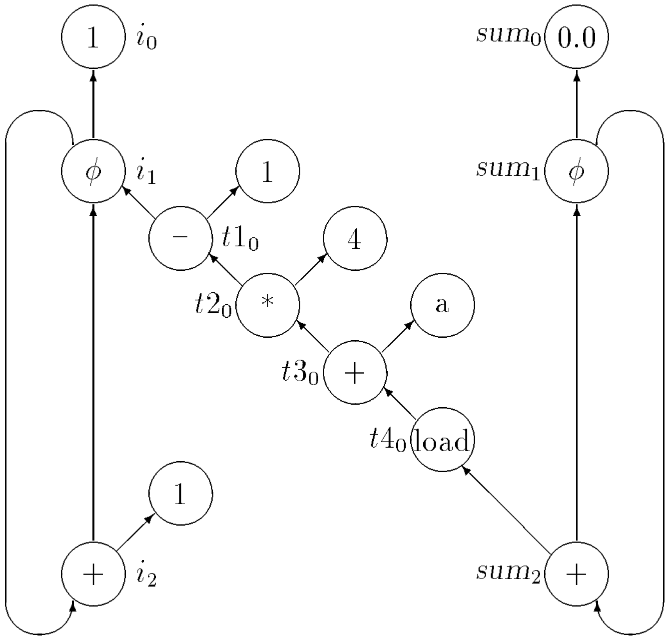


Figure A.3 SSA graph

DFS($node$)
  $node.DFSnum \leftarrow nextDFSnum{+}{+}$
  $node.visited \leftarrow$ `TRUE`
  $node.low \leftarrow node.DFSnum$
  PUSH($node$)
  for each $o \in$ {operands of$node$}
    if not $o.visited$
      DFS(o)
      $node.low \leftarrow$ MIN($node.low, o.low$)
    if $o.DFSnum < node.DFSnum$ and $o \in stack$
      $node.low \leftarrow$ MIN($o.DFSnum, node.low$)
  if $node.low = node.DFSnum$
    $SCC \leftarrow \emptyset$
    do
      $x \leftarrow$ POP()
      $SCC \leftarrow SCC \cup \{x\}$
    while $x \neq node$
    ProcessSCC(SCC)

Figure A.4 Tarjan's SCC finding algorithm

```
while there is an unvisited node n
    DFS(n) (see Figure A.4)


ProcessSCC(SCC)
    if SCC has a single member n
        if n is of the form x ← iv × rc, x ← rc × iv, x ← iv ± rc, or x ← rc + iv
            Replace(n, iv, rc)
        else
            n.header ← NULL
    else
        ClassifyIV(SCC)


RegionConst(name, header)
    return name.op = LOAD_IMMEDIATE or name.block ≫ header


ClassifyIV(SCC)
    for each n ∈ SCC
        if header→RPOnum > n.block→RPOnum
            header ← n.block
    for each n ∈ SCC
        if n.op ∉ {φ, +, −, COPY}
            SCC is not an induction variable
        else
            for each o ∈ {operands of n}
                if o ∉ SCC and not RegionConst(o, header)
                    SCC is not an induction variable
    if SCC is an induction variable
        for each n ∈ SCC
            n.header ← header
    else
        for each n ∈ SCC
            if n is of the form x ← iv × rc, x ← rc × iv, x ← iv ± rc, or x ← rc + iv
                Replace(n, iv, rc)
            else
                n.header ← NULL
```

Figure A.5 Operator strength reduction algorithm this observation and process the nodes as they are popped from the stack. In our method, the SCC-finder drives the entire strength reduction process.

Figure A.5 shows the algorithm for identifying induction variables. As each SCC is identified, the algorithm decides immediately if it represents an induction variable. The test is simple and efficient. The first step is to consider all basic blocks containing a definition inside the SCC and to identify the *header block* – the block with the smallest reverse-postorder number. We use this information to identify region constants with respect to this SCC: a region constant must either be a compile-time constant or its definition must strictly dominate the header block. The RegionConst function in Figure A.5 implements the test to determine if an SSA name is a region constant with respect to a header block.[19] The next step is to check that each operation and ϕ-node in the SCC has the proper form. For ϕ-nodes, each argument must be either a member of the SCC or a region constant. For addition operations, one operand must be a member of the SCC and the other operand must be a region constant. For subtraction operations, the left operand must be a member of the SCC and the right operand must be a region constant. The only other permissible operation is a copy. If the SCC is determined to be an induction variable, we label each node with a pointer to the header block. Otherwise, we check if any of the members are candidates for reduction.

For each node that is not a member of an SCC, we can decide immediately if it is a candidate for strength reduction and perform the code replacement described in the next section. This is an improvement over previous algorithms that require a worklist of candidate instructions. For simplicity, we will restrict our discussion to instructions of the following forms:

$$x \leftarrow i \times j \quad x \leftarrow j \times i \quad x \leftarrow i \pm j \quad x \leftarrow j + i$$

where i is an induction variable and j is a region constant with respect to the header block for i. Allen, Cocke, and Kennedy describe a variety of other candidate types [3]. These are straightforward extensions to the technique. If this operation is a candidate for reduction, the Replace function described in the next section is invoked immediately. Since this function transforms x into an induction variable, x is labeled as an induction variable with the same header block as i. This allows further reduction of operations using x.

---

[19]We use the notation $B_1 \gg B_2$ to signify that B_1 strictly dominates B_2.

A.1.3 Code Replacement

Once we have found a candidate instruction of the form $x \leftarrow i \times j$, we update the code so that the multiply is no longer performed inside the loop. The compiler creates a new SCC in the SSA graph and replaces the instruction with a copy from the node representing the value of $i \times j$. This process is handled by three mutually recursive functions (shown in Figure A.6):

Replace Replace the current operation with a copy from its reduced counterpart.

Reduce Insert code to strength reduce an induction variable and return the SSA name of the result.

Apply Insert an instruction to apply an opcode to two operands and return the SSA name of the result. Simplifications such as constant folding are performed if possible.

The replacement process is supported by a hash table that tracks the results of reduction [28]. This prevents us from performing the same reduction twice and causes the recursion in Reduce to terminate. Access to the hash table is through two functions:

`search` takes an expression (an opcode and two operands) and returns the SSA name of the result.

`add` takes an expression and the name of its result and adds an entry to the table.

The Replace function is straightforward. It provides the top-level call to the recursive function Reduce and replaces the current operation with a copy. The resulting operation must be an induction variable.

The Reduce function is responsible for adding the appropriate operations to the procedure. The first step is to check the hash table for the desired result. If the result is already in the hash table, then no additional instructions are needed, and Reduce returns the SSA name of the result. Otherwise, it must copy the operation or ϕ-node that defines the induction variable and assign a new name to the result. The *copyDef* function does this. Next, Reduce considers each argument of the copy. If the argument is defined inside the SCC, Reduce invokes itself recursively on that argument. Arguments defined outside the SCC are either the initial value of the induction variable or the value by which the induction value is incremented. The initial value must be an argument of a ϕ-node, and the increment value must be an

```
Replace(node, iv, rc)
    result ← Reduce(node.op, iv, rc)
    Replace node with a copy from result
    node.header ← iv.header


SSAname Reduce(opcode, iv, rc)
    result ← search(opcode, iv, rc)
    if result is not found
        result ← inventName()
        add(opcode, iv, rc, result)
        newDef ← copyDef(iv, result)
        newDef.header ← iv.header
        for each operand o of newDef
            if o.header = iv.header
                Replace o with Reduce(opcode, o, rc)
            else if opcode = × or newDef.op = φ
                Replace o with Apply(opcode, o, rc)
    return result


SSAname Apply(opcode, op1, op2)
    result ← search(opcode, op1, op2)
    if result is not found
        if op1.header ≠ NULL and RegionConst(op2, op1.header)
            result ← Reduce(opcode, op1, op2)
        else if op2.header ≠ NULL and RegionConst(op1, op2.header)
            result ← Reduce(opcode, op2, op1)
        else
            result ← inventName()
            add(opcode, op1, op2, result)
            Choose the location where the operation will be inserted
            Decide if constant folding is possible
            Create newOper at the desired location
            newOper.header ← NULL
    return result
```

Figure A.6 Code replacement functions operand of an instruction. The reduction is always applied to the initial value, but the reduction is only applied to the increment if we are reducing a multiply. In other words, when the candidate is an add or subtract instruction, we modify only the initial value, but if the candidate is a multiply, we modify both the initial value and the increment. Therefore, Reduce invokes Apply on arguments defined outside the SCC only if we are reducing a multiply or if we are processing the arguments of a ϕ-node.

The Apply function is conceptually simple, although there are a few details that must be considered. The basic function of Apply is to create an operation with the desired result. We rely on the hash table to determine if such an operation already exists. It is possible that the operation we are about to create is a candidate for strength reduction. If so, we perform the reduction immediately by calling Reduce. This case often arises from triangular loops – where the bounds of an inner loop are a function of the index of an outer loop.

Before inserting the operation, the algorithm must select the location where it is legal to do so. Rather than construct "landing pads" before the loop being reduced, our algorithm relies on dominance information created during SSA construction. Intuitively, the instruction must go into a block that is dominated by both operands. If one of the operands is a constant, we may have to copy its definition to satisfy this condition. Otherwise, both operands must be region constants, so their definitions dominate the header block. One operand must be a descendant of the other in the dominator tree, so the operation can be inserted immediately after the descendant. This avoids the need for landing pads; it may place the operation in a less deeply nested location than the landing pad.

A.1.4 Example

As an example of how the code replacement functions operate, we will apply them to the SSA graph in Figure A.3. The first candidate identified is $t1_0$. We invoke Replace with $iv = i_1$ and $rc = 1$. The hash table search in Reduce will fail, so the first SSA name invented will be osr_0. We add this entry to the hash table and create a copy of the ϕ-node for i_1. Next, we process the arguments of the copy. Since the first argument, i_0, is a region constant, we replace it with the result of Apply, which will perform constant folding and return the SSA name slosr_1. The second argument, i_2, is an induction variable, so we recursively invoke Reduce. Since no match is found

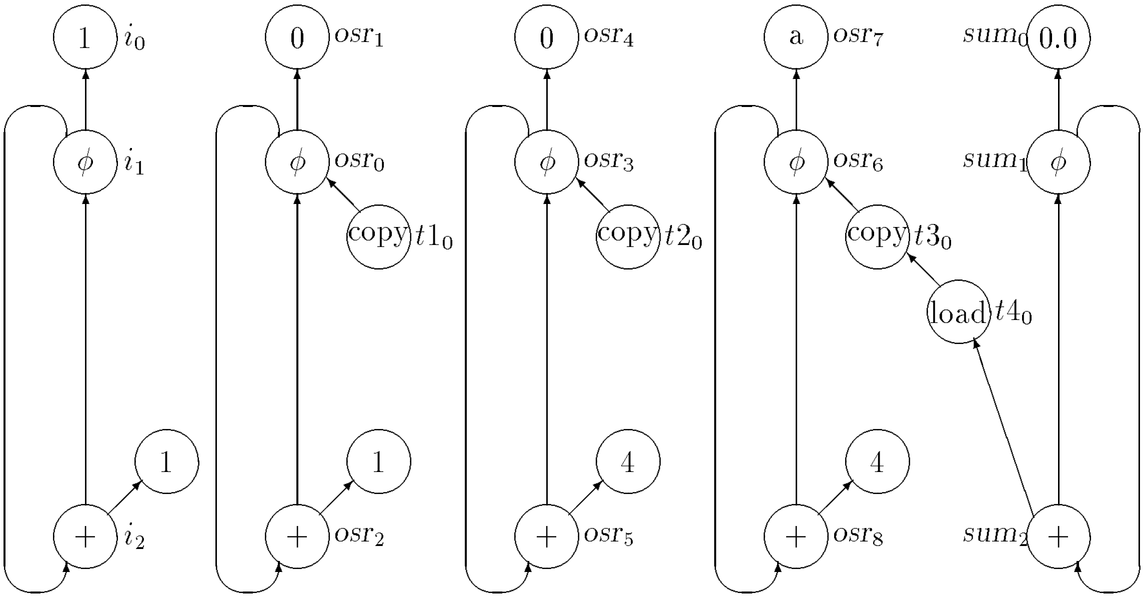


Figure A.7 After operator strength reduction

in the hash table, we invent a new SSA name, slosr_2, add an entry to the hash table, and copy the operation for i_2. The first argument is the region constant 1, which will be left unchanged. The second argument is i_1, which is an induction variable. The recursive call to Reduce will produce a match in the hash table with slosr_0 as the result. At this point, the calls to Reduce finish, and the SSA name osr_0 is returned to Replace. We replace the operation defining $t1_0$ with a copy from osr_0. We label $t1_0$ as an induction variable so that $t2_0$ and $t3_0$ will also be identified as candidates for strength reduction. Figure A.7 shows the SSA graph of our example program after operator strength reduction is completed.

A.1.5 Running Time

The time required to identify the induction variables and region constants in an SSA graph is $\mathbf{O}(N+E)$, where N is the number of nodes and E is the number of edges. The Replace function performs work that is proportional to the size of the SCC containing the induction variable, which can be as large as $\mathbf{O}(N)$. Since Replace can be invoked $\mathbf{O}(N)$ times, the worst case running time is $\mathbf{O}(N^2)$. This seems expensive; unfortunately, it is necessary. Figure A.8 shows a program that generates this worst

$i \leftarrow 0$

`while` (P_0) `do`
  `if` (P_1) `then`
    $i \leftarrow i + 1$
    $k \leftarrow i \times c_1$
  `if` (P_2) `then`
    $i \leftarrow i + 2$
    $k \leftarrow i \times c_2$
  $\cdots$
  `if` (P_n) `then`
    $i \leftarrow i + n$
    $k \leftarrow i \times c_n$
`end`

Original code

$i \leftarrow 0$ $t_2 \leftarrow 0$
$t_1 \leftarrow 0$ $\cdots$
$t_n \leftarrow 0$
`while` (P_0) `do`
  `if` (P_1) `then`
    $t_1 \leftarrow t_1 + c_1$ $t_n \leftarrow t_n + c_n$
    $t_2 \leftarrow t_2 + c_2$ $i \leftarrow i + 1$
    $\cdots$ $k \leftarrow t_1$
  `if` (P_2) `then`
    $t_1 \leftarrow t_1 + 2 \times c_1$ $t_n \leftarrow t_n + 2 \times c_n$
    $t_2 \leftarrow t_2 + 2 \times c_2$ $i \leftarrow i + 2$
    $\cdots$ $k \leftarrow t_2$
  $\cdots$
  `if` (P_n) `then`
    $t_1 \leftarrow t_1 + n \times c_1$ $t_n \leftarrow t_n + n \times c_n$
    $t_2 \leftarrow t_2 + n \times c_2$ $i \leftarrow i + n$
    $\cdots$ $k \leftarrow t_n$
`end`

Transformed code

Figure A.8 A worst-case example

case behavior in the replacement step. It requires introduction of a quadratic number of updates. Note that this behavior is a function of the code being transformed, not any particular details of our algorithm. Any algorithm that performs strength reduction on this code will have this behavior. In practice, experience with strength reduction suggests that this problem does not arise. In fact, we have not seen this problem mentioned in the literature. Since the amount of work is proportional to the number of instructions inserted, any algorithm for strength reduction that reduces these cases will have the same, or worse, complexity.

A.2 Linear Function Test Replacement

After strength reduction, the code often contains induction variables whose sole use is to govern control flow. If so, linear function test replacement can convert them into dead code. The compiler should look for comparisons between an induction variable and a region constant. For example, the comparison "**if** ($i_2 \leq 100$) **goto** L" in our example program (see Figure A.1) could be replaced with "**if** ($osr_8 \leq 396{+}a$) **goto** L". This transformation is called *linear function test replacement* (LFTR).

Previous methods would search the hash table for an expression containing the induction variable referenced in the comparison. In the example in Figures A.3 and A.7, a "chain" of reductions was applied to node i_2. If LFTR is to be effective, we must follow the entire chain quickly. To facilitate this process, Reduce records the reductions it performs on each node in the SSA graph. Each reduction is represented by an edge from a node to its strength-reduced counterpart labeled with the opcode and the region constant of the reduction. When a candidate for LFTR is found, it is a simple matter of traversing these edges, inserting code to compute the new region constant for the test, and updating the compare instruction. We use two procedures to support this process:

FOLLOW_EDGES Follow the LFTR edges and return the SSA name of the last one in the chain.

ApplyEdges Apply the operations represented by the LFTR edges to a region constant and return the SSA name of the result.

The ApplyEdges function can be easily implemented using the Apply function described in Section A.1.3. For each LFTR candidate, we replace the induction variable with the result of FOLLOW_EDGES, and we replace the region constant with the result of ApplyEdges. In the example in Figure A.2, there are two sets of these edges:

- $i_1 \xrightarrow{-1} osr_0 \xrightarrow{\times 4} osr_3 \xrightarrow{+a} osr_6$
- $i_2 \xrightarrow{-1} osr_2 \xrightarrow{\times 4} osr_5 \xrightarrow{+a} osr_8$

To transform the test $i_2 \leq 100$, we replace i_2 with the result of FOLLOW_EDGES, osr_8, and we replace 100 with the result of ApplyEdges, $(((100-1)\times 4)+a) = 396+a$. Notice that LFTR renders the original induction variable dead. Subsequent optimizations should remove the associated instructions.

A.2.1 Follow-up Transformations

The algorithm presented here operates in a compiler that performs a suite of optimization passes. To provide a good separation of concerns, we leave much of the "cleaning up" to other well-known optimizations that should be run after operator strength reduction.

Since operator strength reduction has the potential to introduce equal induction variables, we need either value partitioning or SCC-based value numbering (see Chapters 3 and 4) to detect these equalities. Hash-based value numbering (see Chapter 2 will be unable to detect the equality of SCCs because they contain values flowing through back edges.

The SSA graph in Figure A.7 contains a great deal of dead code. This is because many of the use-definition edges in the original SSA graph have been changed, resulting in "orphaned" nodes. We rely on a separate pass of *dead code elimination* to remove these instructions [18, Section 7.1].

Many of the copies introduced during strength reduction can be eliminated. For example, the copy into $t3_0$ in Figure A.7 can be eliminated if the load into $t4_0$ uses the value of osr_6 directly. We rely on the *copy coalescing* phase of a Chaitin-style graph coloring register allocator to accomplish this task [11, 7].

A.3 Previous Work

Reduction of operator strength has a long history in the literature. The classic method is presented in a paper by Allen, Cocke, and Kennedy [3]. It, in turn, builds on earlier work by Cocke and Kennedy [15, 28]. These algorithms transform one loop at a time, working outward through each loop nest, making passes to generate def-use chains, find loops and insert landing pads, find region constants and induction variables, and to perform the actual reduction and instruction replacement. Linear function test replacement is a separate pass for each loop. Chase extended the Allen, Cocke, and Kennedy method to reduce more additions [12].

A second family of techniques has grown up around the literature of data-flow analysis [19, 26, 20, 30]. These methods use the careful code placement calculations developed for code motion to perform strength reduction. These methods avoid the control-flow analysis used in the Allen, Cocke, and Kennedy methods; our algorithm uses properties of SSA for the same purpose. Their placement techniques avoid lengthening execution paths; our algorithm cannot make the same claim. Their principal limitation is that they work from a simpler notion of a region constant—only literal constants can be found. The Allen, Cocke, and Kennedy-style techniques, including ours, include loop invariant values as region constants.

Paige has looked at reducing a number of set operators [36, 35] and using multiset discrimination as an alternative to hashing to avoid its worst case behavior [8]. Sites looked at the related issue of minimizing the number of loop induction variables [41]. Markstein, Markstein, and Zadeck, in a chapter for a forthcoming ACM Press Book, present an algorithm that combines strength reduction, expression reassociation, and code motion. Our work has treated these issues separately.

A.4 Summary

This chapter presents a simple and elegant algorithm for performing reduction of operator strength. The results of applying our method are similar to those achieved by the Allen, Cocke, and Kennedy algorithm. Our technique relies on prior optimizations and properties of the SSA graph to produce an algorithm that (1) is simple to understand and to implement, (2) relies on the dominator tree which must be computed during SSA construction rather than the loop structure of the program (which can be costly to compute), (3) avoids instantiating the sets of induction variables and region constants required by other algorithms, (4) processes each candidate instruction immediately rather than maintaining a worklist, and (5) greatly simplifies linear function test replacement. The result is an efficient algorithm that is both easy to understand and easy to implement.

Bibliography

[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] 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.

[4] 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.

[5] 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.*

[6] Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. *SIGPLAN Notices*, 27(7):311–321, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.*

[7] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.

[8] 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.

[9] David Callahan, Steve Carr, and Ken Kennedy. Improving register allocation for subscripted variables. *SIGPLAN Notices*, 25(6):53–65, June 1990. *Proceedings*

of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation.

[10] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. Compiler optimizations for improving data locality. In Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 252–262, San Jose, California, 1994.

[11] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, January 1981.

[12] David R. Chase. Brief survey of optimizations. Unpublished paper, 1987.

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

[14] John Cocke. Global common subexpression elimination. *SIGPLAN Notices*, 5(7):20–24, July 1970. *Proceedings of a Symposium on Compiler Optimization.*

[15] John Cocke and Ken Kennedy. An algorithm for reduction of operator strength. *Communications of the ACM*, 20(11), November 1977.

[16] John Cocke and Peter Markstein. Measurement of program improvement algorithms. In *Proceedings of Information Processing 80.* North Holland Publishing Company, 1980.

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

[18] 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.

[19] Dhananjay M. Dhamdhere. On algorithms for operator strength reduction. *Communications of the ACM*, pages 311–312, May 1979.

[20] Dhananjay M. Dhamdhere. A new algorithm for composite hoisting and strength reduction. *International Journal of Computer Mathematics*, pages 1–14, 1989.

[21] Karl-Heinz Drechsler and Manfred P. Stadel. A solution to a problem with Morel and Renvoise's "Global optimization by suppression of partial redundancies". *ACM Transactions on Programming Languages and Systems*, 10(4):635–640, October 1988.

[22] 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.

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

[24] Particia C. Goldberg. A comparison of certain optimization techniques. In Rustin, editor, *Design and Optimization of Compilers*, pages 31–50. Prentice-Hall, 1972.

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

[26] J.R. Issac and Dhananjay M. Dhamdhere. A composite algorithm for strength reduction and code movement. *International Journal of Computer and Information Sciences*, pages 243–273, 1980.

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

[28] Ken Kennedy. Reduction in strength using hashed temporaries. SETL Newsletter 102, Courant Institute of Mathematical Sciences, New York University, March 1973.

[29] 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.*

[30] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy strength reduction. *Journal of Programming Languages*, 1(1):71–91, 1993.

[31] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, July 1994.

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

[33] 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, July 1979.

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

[35] Robert Paige and Shaye Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, July 1982.

[36] Robert Paige and Jacob T. Schwartz. Reduction in strength of high level operations. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 58–71, Los Angeles, California, January 1977.

[37] Gordon D. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[38] Lori L. Pollock. *An Approach to Incremental Compilation of Optimized Code*. PhD thesis, University of Pittsburgh, 1986.

[39] John H. Reif. Symbolic programming analysis in almost linear time. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 76–83, Tucson, Arizona, January 1978.

[40] Vatsa Santhanam. Register reassociation in PA-RISC compilers. *Hewlett-Packard Journal*, pages 33–38, June 1992.

[41] Richard L. Sites. The compilation of loop induction expressions. *ACM Transactions on Programming Languages and Systems*, 1(1):50–57, July 1979.

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

[43] Robert Endre Tarjan. Testing flow graph reducibility. *Journal of Computer and System Sciences*, 9:355–365, 1974.

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

[45] 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.

[46] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. *SIGPLAN Notices*, 26(6):30–44, June 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.*

[47] Michael Wolfe. Beyond induction variables. *SIGPLAN Notices*, 27(7):162–174, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.*