Value-Driven Code Motion

Keith Cooper
Taylor Simpson

CRPC-TR95637-S
October 1995

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

Value-Driven Code Motion

Keith D. Cooper
L. Taylor Simpson

Value-driven code motion 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 have not exploited it. By understanding how code motion interacts with global value numbering, we can simplify and improve the code motion framework. 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 paper focuses on lazy code motion as proposed by Knoop, Rüthing, and Steffen and modified by Drechsler and Stadel [14, 15, 11]. That algorithm is provably optimal; this paper 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 present experimental data that shows both these effects.

1 Introduction

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. One problem with this approach is that critical information is lost between the 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 equivalent.
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. In other words, we can select the algorithm for step one independently of the selection of the algorithm for step two. The results of step one are communicated to step two by assigning a unique name (value number) to each value and rewriting the program accordingly. This paper describes how to take advantage of this information to improve code motion. Section 2 explains what information must be communicated by the value numbering step and reviews several techniques for discovering it. Section 3 describes previous techniques for redundancy elimination. Section 4 presents our approach to value-driven redundancy elimination, and Section 5 shows experimental results demonstrating that our technique is indeed an improvement over previous ones.

2 Value Numbering

Value numbering is a code optimization 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

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

collection of optimizations that vary in power and scope. All of the algorithms discussed here operate on the static single assignment (SSA) form of the code [9]. The primary objective is to assign an identifying number (a *value number*) to each value computed in the routine in a way that two values have the same number if the compiler can prove they are equal for all possible program inputs.

In value numbering, the compiler can only assign two expressions the same number if it can prove that they always produce equal values. We will discuss three techniques for proving this equivalence. In general, each one 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.

2.1 Hash-Based Value Numbering

Cocke and Schwartz describe a local technique that uses hashing to discover redundant computations and fold constants [7]. Each unique value is identified by its *value number*. Two computations have the same value number if they are provably equal. In the literature, this technique and its derivatives are called "value numbering."

Briggs *et al.* introduced an extension that performs hash-based value numbering over an entire procedure by using a single hash table for all the basic blocks [6]. It achieves a consistent name space for the routine by replacing every SSA name with its value number. This form of value numbering cannot remove instructions as they are processed because the relative locations of two equal computations are not known. In contrast, when value numbering over a single basic block, an expression exists in the hash table only when it is computed earlier in the same basic block.

In this method, the compiler traverses the control-flow graph (CFG) in reverse postorder and process the ϕ-nodes and instructions in each block. Before it can process the ϕ-nodes in a block, it must check that there are no incoming back edges. If a ϕ-node references a value that flows through a back edge, that parameter may not have a value number yet. If so, the compiler must assign a unique value number to each ϕ-node in the block. If no back edges flow into a block, the reverse-postorder traversal guarantees that all ϕ-node parameters have been assigned value numbers. If so, the ϕ-nodes in the block can be analyzed.

When processing the ϕ-nodes and instructions in a block, the algorithm overwrites the operands with their value numbers, folds constants, simplifies algebraic identities, and searches the table for the resulting expression. If the expression is found, the result is overwritten with the value number of the expression. Otherwise, the expression is added to the table with the SSA name defined by the ϕ-node or instruction as its value number.

Figure 1 demonstrates how this technique works. Both computations of the expression $a + b$ are assigned the name x_0 and both computations of $c + d$ are assigned z_0. Notice that even though the expression $c + d$ exists in the hash table when block C is processed, it would be unsafe to remove the instruction. The ϕ-node in block D is removed because all of its parameters are equal.

Technically, this is not a global algorithm because it cannot handle values that travel through back edges in the CFG. However, it does name values consistently throughout the routine, and this is the key feature that we require for value-driven code motion. While other value numbering algorithms will discover a different set of equivalences, our experiments show that this algorithm is competitive in practice with the global algorithms presented in Sections 2.2 and 2.3 [6].

2.2 Value Partitioning

Alpern, Wegman, and Zadeck present a technique that uses a variation on Hopcroft's DFA-minimization algorithm to partition values into congruence classes [4, 1]. It operates on the SSA form of the routine [9]. Two values are *congruent* if they are computed by the same opcode, and their corresponding operands are

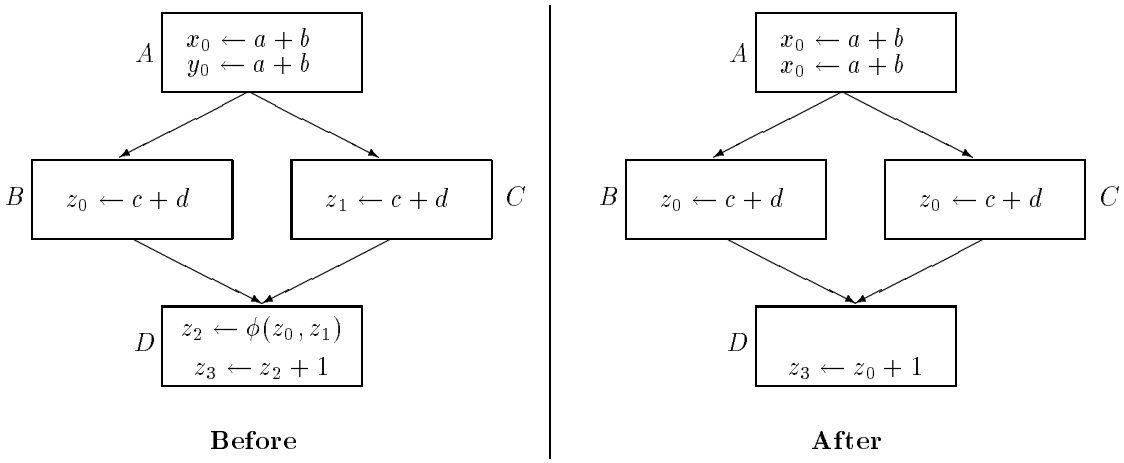

Figure 1 Hash-based value numbering

congruent. For all legal expressions, two congruent values must be equal. Since the definition of congruence is recursive, there will be routines where the solution is not unique. A trivial solution would be to set each value in the routine to be congruent only to itself; however, the solution we seek is the *maximal fixed point* – the solution that contains the most congruent values.

Initially, the partition contains a congruence class for the values defined by each operator in the program. The partition is iteratively refined by examining the uses of all members of a class and determining which classes must be further subdivided. After the partition stabilizes, the registers and ϕ-nodes in the routine are renumbered based on the congruence classes. Because the effects of partitioning and renumbering are analogous to those of value numbering described in the previous section, we think of this technique as a form of global (or intraprocedural) value numbering.[1] The drawback of value partitioning is that it cannot handle algebraic identities such as the ϕ-node eliminated by hash-based value numbering in Figure 1.

2.3 SCC-Based Value Numbering

SCC-based value numbering can discover at least as many equivalences as hash-based value numbering or value partitioning [8]. The algorithm operates in conjunction with Tarjan's algorithm for finding strongly connected components (SCCs) [18]. 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 [13]. The loop connectedness of a graph is the maximum number of back edges in any acyclic path. This number can be as large as $\mathbf{O}(N)$, but it is believed that, in practice, this number is bounded by a small constant. The running time of the iterative data-flow analysis used in code motion is $\mathbf{O}(N \times B \times D(\text{CFG}))$, where B is the number of blocks in the CFG and $D(\text{CFG})$ is the loop connectedness of the CFG.

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. Therefore, we assign value numbers as nodes are popped from the stack. When a single node is popped from the stack, we know that we have assigned value numbers to the operands of the corresponding expression. Thus, we can examine the expression and assign a value number to this node. When a collection of nodes representing an SCC is

[1] Rosen, Wegman, and Zadeck describe a technique called *global value numbering* [17]. It is an interesting and powerful approach to redundancy elimination, but it should not be confused with value partitioning.

popped, we know that we have assigned value numbers to any operands outside the SCC. The members of the SCC require special handling in order to perform value numbering.

We assign value numbers to the nodes of an SCC by iterating in reverse postorder. Initially, the value number for each member of the SCC is $\top$ (pronounced "top"). A value number of $\top$ indicates that this value has not yet been examined. In order to make optimistic assumptions about the values and later disprove them, we use two hash tables. The iterative phase uses an *optimistic* table. Once the value numbers in the SCC stabilize, entries are added to the *valid* table. The value numbering of single values uses only the valid table.

2.4 Summary and Comparison

The key difference between these algorithms is the set of equivalences they discover. Hash-based value numbering can discover constant values, evaluate expressions whose operands are constants, and propagate these values through the code. It can also recognize certain algebraic identities, like $i = i + 0$ and $j = j \times 1$, and use them to simplify the code and to expand the set of values known to be equal. Value partitioning does not propagate constants or simplify identities, but it can analyze values that flow through back edges in the CFG. SCC-based value numbering combines the best features of hash-based value numbering and value partitioning. It can propagate constants, simplify identities, and analyze values that flow through back edges.

3 Code Removal and Motion

Value numbering, as described in Section 2, rewrites the routine so that each name corresponds to a unique value number. Therefore, better value numbering results in fewer names. However, value numbering alone will not improve the running time of the routine; the compiler must also find and remove redundant computations. This is accomplished either by eliminating instructions or by moving instructions to less frequently executed locations. This section describes previous approaches to code removal and motion.

3.1 Available Expressions

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.

Properties of value numbering let us simplify the formulation of AVAIL. The traditional data-flow equations deal with lexical *names*, while our equations deal with *values*. 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 on the left side of Figure 2. Under the traditional data-flow framework, the assignment to X would "kill" the Z expression. However, if the assignment to X caused

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

Example

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

Data-Flow Equations

Figure 2 AVAIL-Based Removal

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 2. Notice that the equation for AVOUT_i does not include a term for the expressions killed in block i. We simply add the set of values defined in the block (defined_i) to the set of values available at the beginning of the block (AVIN_i). The set defined_b is the set of value numbers with a definition in block b. This is a *superset* of the set of values generated in b (gen_b) which does not include any expressions whose definition is followed by a modification of one of its subexpressions.

3.2 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.

Partial redundancy elimination (PRE) is an optimization introduced by Morel and Renvoise that combines common subexpression elimination with loop invariant code motion [16, 10]. Partially redundant computations are redundant along some, but not necessarily all, execution paths. One interesting property of PRE is that is never lengthens a path through the program.

Knoop, Rüthing, and Steffen describe a descendant of PRE, called lazy code motion (LCM) [14, 15]. Their technique avoids the unnecessary code motion inherent in PRE. This feature is important when code motion interacts with register allocation and other optimizations. Drechsler and Stadel present a variation of this technique that they claim is more practical [11]. Although our value-driven approach could be applied to any of these data-flow frameworks, we will show specifically how to extend Drechsler and Stadel's framework.

4 Value-Driven Code Motion

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 research is to 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 3.1. 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.

In value-driven code motion (VDCM), certain operations cannot be moved; they are called *fixed*. Examples of fixed operations include ϕ-nodes, subroutine calls, and control-flow operations. All other operations are considered *candidates* for code motion.[2]

The first step of VDCM is to find available expressions as described in Section 3.1. The only local predicate required for this framework is 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 altered_b represents the set of values that cannot move past some definition in block b.[3] In

[2] This same distinction exists in both PRE and LCM.

[3] The analogous predicate in Drechsler and Stadel is $\overline{\text{TRANSP}_b}$.

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

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

Anticipatability

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

Earliest

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

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

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 3 Data-flow equations for code motion

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.

Intuitively, 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, altered_b can be computed more efficiently by starting with $\text{AVOUT}_b - \text{AVIN}_b$, 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.

Compare this approach with the technique for computing the altered set using lexical names. 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 names. Finally, for each block b, the analyzer computes altered_b as the union of the dependences for any fixed value defined in b.

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.,* $\text{defined}_b - \text{altered}_b$).

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.,* $\text{INSERT}_{i,j}$ for each edge $e = (i, j)$) and the set of values to delete from each block (*i.e.,* DELETE_b for each block b). For reference, the complete set of equations is given in Figure 3.

The example in Figure 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.,* $\text{altered}_{B_2} = \{r_1, r_2\}$). 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 2. The solution to these equations shows that r_1 is available on entry and exit for block B_2 ($r_1 \in \text{AVIN}_{B_2}$ and $r_1 \in \text{AVOUT}_{B_2}$). We initialize the set altered_{B_2} with $\text{AVOUT}_{B_2} - \text{AVIN}_{B_2}$. Since $r_1 \notin \text{AVOUT}_{B_2} - \text{AVIN}_{B_2}$, $r_2 \notin \text{altered}_{B_2}$, 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.

5 Experimental Results

Even though LCM is provably optimal, we have shown that assumptions about the input program can be modified. To assess the impact of VDCM over LCM, we have implemented both of the optimizations in our

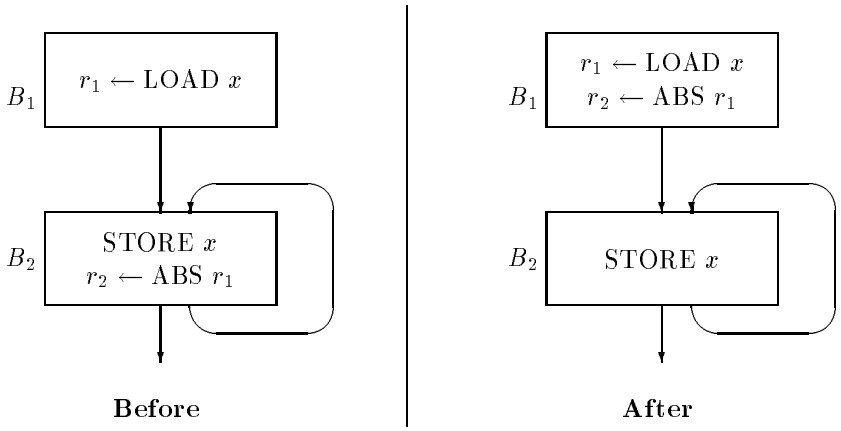

Figure 4 Example

experimental Fortran compiler. Comparisons were made using routines from a suite of benchmarks consisting of over 50 routines drawn from the SPEC benchmark suite and from Fosythe, Malcolm, and Moler's book on numerical methods [12].

Our optimizer is composed of a sequence of passes that operate on ILOC – our intermediate language. ILOC is a pseudo-assembly language for a RISC machine with an arbitrary number of symbolic registers. The back end generates code that is capable of counting the number of ILOC operations executed. Routines are optimized using the sequence of global reassociation [5], SCC-based value numbering [8], code motion (the type is indicated in the legend), global constant propagation [19], operator strength reduction [3], SCC-based value numbering, global constant propagation, global peephole optimization, dead code elimination [9, Section 7.1], copy coalescing, and a pass to eliminate empty basic blocks. We repeat the value numbering and constant propagation passes to clean up after operator strength reduction. Figures 5 and 6 show only those routines where there was variation in the number of ILOC operations executed. Each column represents dynamic counts of ILOC operations, normalized against LCM. Recall that LCM is an optimal method; thus, we expect the improvements in code motion to be relatively minor. In no case did VDCM do worse than LCM.

We also compared the time required by each of the techniques for some of the larger routines in the test suite. These results are shown in Table 1. The number of blocks, SSA names, and operations are given to indicate the size of the routine being optimized. 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

				LCM			VDCM		
routine	*blocks*	*SSA names*	*operations*	*set size*	*altered*	*total*	*set size*	*altered*	*total*
`tomcatv`	131	2212	2663	790	0.41	0.52	924	0.06	0.18
`ddeflu`	109	5494	4502	1420	1.67	1.82	3632	0.10	0.49
`debflu`	116	5856	3951	1073	0.75	0.88	3851	0.09	0.52
`deseco`	251	13164	11771	2500	4.32	4.90	6054	0.43	1.78
`twldrv`	266	23486	15615	3792	9.07	9.95	16004	0.83	4.69

Table 1 Running times of code motion techniques

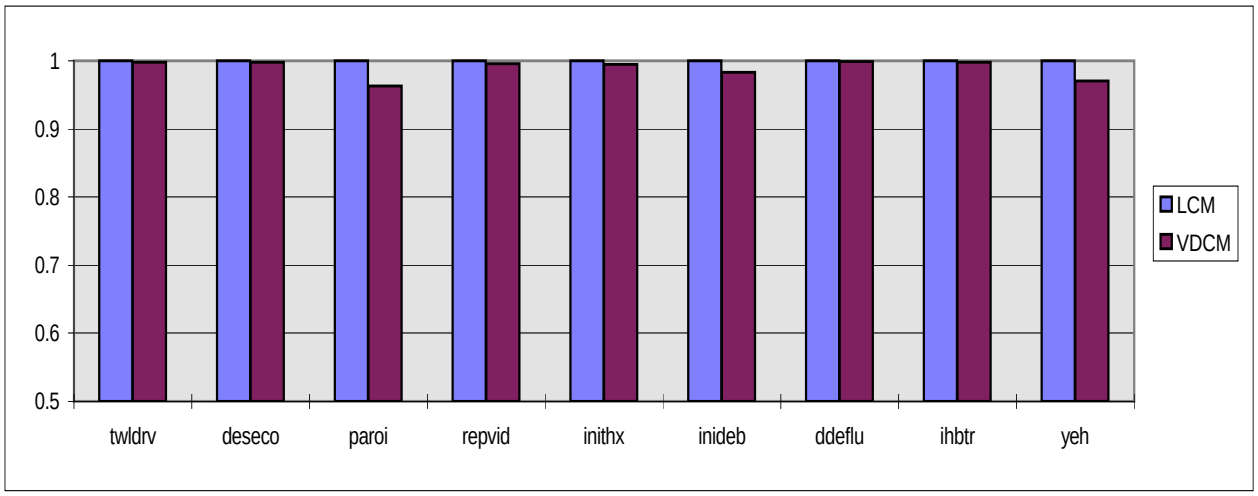

Figure 5 Comparison of code motion techniques – SPEC benchmark

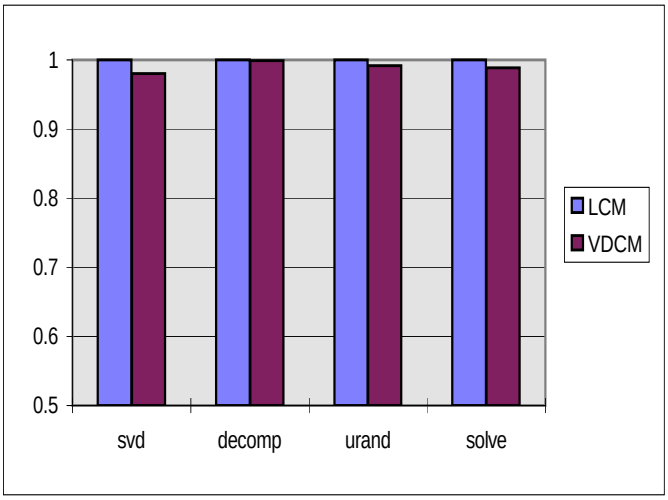

Figure 6 Comparison of code motion techniques – FMM benchmark

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.

6 Conclusion

We have presented 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 have applied 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. Further, our algorithm runs faster than lazy code motion because the computation of the altered set for each block is greatly simplified. We experimentally compared the improvements made by our technique over lazy code motion when applied to real programs in the context of an optimizing compiler. Finally, we showed that our optimization runs significantly faster than lazy code motion.

7 Acknowledgements

Our colleagues in the Massively Scalar Compiler Project at Rice have played a large role in this work. Without their implementation efforts, we could not have completed this work. We would also like to thank Vivek Sarkar and IBM for supporting Taylor Simpson through the IBM Cooperative Fellowship.

References

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

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

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

[8] Keith D. Cooper and L. Taylor Simpson. SCC-based value numbering. Extended abstract submitted to SIGPLAN PLDI '96.

[9] 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.

[10] 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.

[11] 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.

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

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

[14] 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.*

[15] 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.

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

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

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

[19] 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.