Operator Strength Reduction

Keith Cooper
Taylor Simpson
Christopher Vick

CRPC-TR95635-S
October 1995

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

Operator Strength Reduction

Keith D. Cooper
L. Taylor Simpson
Christopher A. Vick

Operator strength reduction is a well-known code improvement technique. It improves compiler-generated code by reformulating certain costly computations in terms of other less expensive ones. A common case arises in array addressing expressions used in loops. The compiler can replace the sequence of multiplies with an equivalent sequence of additions. Together with linear function test replacement, this technique can significantly reduce the execution time of loops containing array references.

This paper presents an algorithm for performing strength reduction on the static single assignment (SSA) form of a procedure [12]. It produces results similar to an earlier algorithm due to Allen, Cocke, and Kennedy [2]. Because it takes advantage of the properties of SSA form, it has several advantages over that technique: (1) it is simple to understand and to implement, (2) it 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) it avoids instantiating the sets of induction variables and region constants required by other algorithms, (4) it processes each candidate instruction immediately rather than maintaining a worklist, and (5) it greatly simplifies linear function test replacement. We have implemented the algorithm in an SSA-based optimizing compiler.

1 Introduction

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 [2].

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 [11, 26, 4], 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 [31, 8].

This paper presents a new algorithm for performing strength reduction. It produces results similar to those of Allen, Cocke, and Kennedy's classic algorithm [2]. By assuming some specific prior optimizations and operating on the SSA form of the procedure [12], 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 construction rather than the loop structure of the program (which can be costly to compute), (3) avoids instantiating the

Authors addresses: Keith Cooper and Taylor Simpson are currently at Rice University, 6100 South Main Street, Mail Stop 41, Houston, TX 77005. Chris Vick is at Texas Instruments, PO Box 1443, Mail Stop 730, Houston, TX 77251-1443. Address all correspondence to Taylor Simpson, `lts@cs.rice.edu`

This research has been supported by ARPA, IBM Corporation, and Texas Instruments Corporation.

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

Source code

```
      $sum \leftarrow 0.0$
      $i \leftarrow 1$
L:    $t1 \leftarrow i - 1$
      $t2 \leftarrow t1 \times 4$
      $t3 \leftarrow t2 + a$
      $t4 \leftarrow$ load $t3$
      $sum \leftarrow sum + t4$
      $i \leftarrow i + 1$
      if ($i \leq 100$) goto $L$
```

Intermediate code

```
      $sum_0 \leftarrow 0.0$
      $i_0 \leftarrow 1$
L:    $sum_1 \leftarrow \phi(sum_0, sum_2)$
      $i_1 \leftarrow \phi(i_0, i_2)$
      $t1_0 \leftarrow i_1 - 1$
      $t2_0 \leftarrow t1_0 \times 4$
      $t3_0 \leftarrow t2_0 + a$
      $t4_0 \leftarrow$ load $t3_0$
      $sum_2 \leftarrow sum_1 + t4_0$
      $i_2 \leftarrow i_1 + 1$
      if ($i_2 \leq 100$) goto $L$
```

SSA form

Figure 1 Example

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 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 2 shows the result of strength reduction and the optimizations described in Section 3.1. 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 "dead"). This transformation is called *linear function test replacement* (LFTR). The results of applying LFTR appear in the right column of Figure 2.

2 The Algorithm

2.1 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.

Our compiler utilizes *lazy code motion* (LCM) to accomplish loop invariant code motion and common subexpression elimination [19, 21, 16]. This technique is a descendant of partial redundancy elimination introduced by Morel and Renvoise [23, 15]. We perform global reassociation and global renaming prior to LCM [4].

Because LCM will not move conditionally executed code out of loops, it is possible that some loop invariant

```
      $sum_0 \leftarrow 0.0$
      $i_0 \leftarrow 1$
      $t5_0 \leftarrow a$
L:    $sum_1 \leftarrow \phi(sum_0, sum_2)$
      $i_1 \leftarrow \phi(i_0, i_2)$
      $t5_1 \leftarrow \phi(t5_0, t5_2)$
      $t4_0 \leftarrow$ load $t5_1$
      $sum_2 \leftarrow sum_1 + t4_0$
      $i_2 \leftarrow i_1 + 1$
      $t5_2 \leftarrow t5_1 + 4$
      if ($i_2 \leq 100$) goto $L$
```

After strength reduction

```
      $sum_0 \leftarrow 0.0$
      $t5_0 \leftarrow a$
L:    $sum_1 \leftarrow \phi(sum_0, sum_2)$
      $t5_1 \leftarrow \phi(t5_0, t5_2)$
      $t4_0 \leftarrow$ load $t5_1$
      $sum_2 \leftarrow sum_1 + t4_0$
      $t5_2 \leftarrow t5_1 + 4$
      if ($t5_2 \leq 396 + a$) goto $L$
```

After linear function test replacement

Figure 2 Transformed Code

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 [30].

We construct the *pruned* SSA *form* of the program [12]. 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 3 shows the SSA graph for the example program in Figure 1.

2.2 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 [29]. 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 [22]. 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 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 [32]. To discover the SCCs, we will use Tarjan's algorithm based on depth-first search [28]. 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.

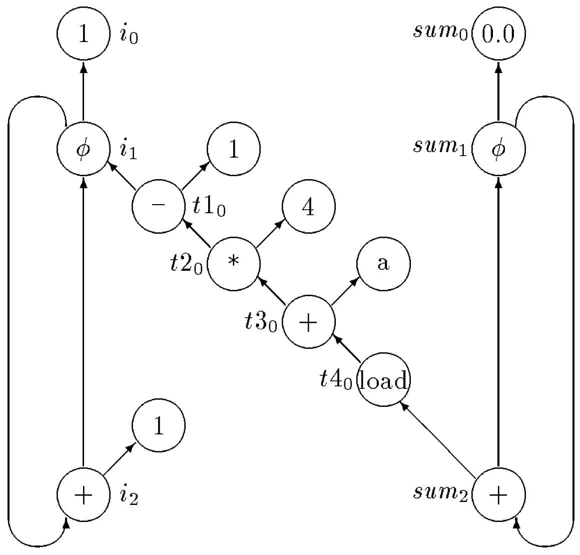

Figure 3 SSA graph

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 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.

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 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.

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:

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

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 [2]. 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.

2.3 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:

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 [18]. 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.

```
void REPLACE(operation, iv, rc)
    result ← REDUCE(operation→opcode, iv, rc)
    Replace operation with a copy from result
    operation→header ← iv→header
```

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. If we are reducing a multiply operation, we apply the operation in both cases. However, if we are reducing an addition operation, we only operate on the initial value. 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.

```
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)
        for each operand o of newDef
            if o is an induction variable
                Replace o with REDUCE(opcode, o, rc)
            else if opcode == × or newDef is a φ-node
                Replace o with APPLY(opcode, o, rc)
    return result
```

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.

```
SSAname APPLY(opcode, op1, op2)
    result ← search(opcode, op1, op2)
    if result is not found
        if op1 is an induction variable and op2 is a region constant
            result ← REDUCE(opcode, op1, op2)
        else if op2 is an induction variable and op1 is a region constant
            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 a new operation at the desired location
    return result
```

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.

2.4 Example

As an example of how the code replacement functions operate, we will apply them to the SSA graph in Figure 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 osr_1. The second argument, i_2, is an induction variable, so we recursively invoke REDUCE. Since no match is found in the hash table, we invent a new SSA name, osr_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 osr_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 4 shows the SSA graph of our example program after operator strength reduction is completed.

2.5 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

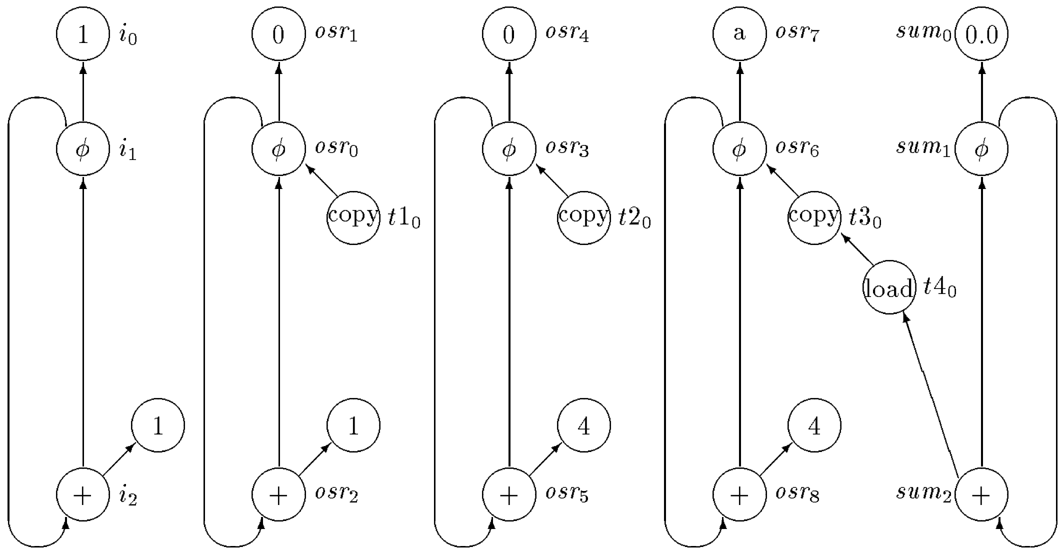

Figure 4 After Operator Strength Reduction

```
          $i = 0$



while ($P_0$) do
     if ($P_1$) then
          $i = i + 1$
          $k = i * c_1$

     if ($P_2$) then
          $i = i + 2$
          $k = i * c_2$

     ...
     if ($P_n$) then
          $i = i + n$
          $k = i * c_n$

end
```

Original code

```
$i = 0$         $t_2 = 0$
$t_1 = 0$       ...
                $t_n = 0$
while ($P_0$) do
     if ($P_1$) then
          $t_1 = t_1 + c_1$              $t_n = t_n + c_n$
          $t_2 = t_2 + c_2$              $i = i + 1$
          ...                            $k = t_1$
     if ($P_2$) then
          $t_1 = t_1 + 2 * c_1$          $t_n = t_n + 2 * c_n$
          $t_2 = t_2 + 2 * c_2$          $i = i + 2$
          ...                            $k = t_2$
     ...
     if ($P_n$) then
          $t_1 = t_1 + n * c_1$          $t_n = t_n + n * c_n$
          $t_2 = t_2 + n * c_2$          $i = i + n$
          ...                            $k = t_n$
end
```

Transformed code

Figure 5 A worst-case example

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 5 shows a program that generates this worst 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.

3 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 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 3 and 4, 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.

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

The APPLY_EDGES function can be easily implemented using the APPLY function described in Section 2.3. 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 APPLY_EDGES. Notice that LFTR renders the original induction variable dead. Subsequent optimizations should remove the associated instructions.

In the example in Figure 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 APPLY_EDGES, $(((100 - 1) \times 4) + a) = 396 + a$.

3.1 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 a *global value numbering* algorithm to detect and remove common subexpressions. Any non-global technique will be unable to detect the equality of SCCs because they contain values flowing through back edges. Alpern, Wegman, and Zadeck presented a technique that uses a variation on Hopcroft's DFA-minimization algorithm to partition values into congruence classes [3, 1]. We have made improvements to this algorithm that include AVAIL-based removal [5].

The SSA graph in Figure 4 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 [12, Section 7.1].

Many of the copies introduced during strength reduction can be eliminated. For example, the copy into $t3_0$ in Figure 4 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 [6].

4 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 [2]. It, in turn, builds on earlier work by Cocke and Kennedy [10, 18]. 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 [9].

A second family of techniques has grown up around the literature of data-flow analysis [13, 17, 14, 20]. 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 [25, 24] and using multiset discrimination as an alternative to hashing to avoid its worst case behavior [7]. Sites looked at the related issue of minimizing the number of loop induction variables [27]. 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.

5 Conclusions

This paper 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.

6 Acknowledgements

Our colleagues on the Massively Scalar Compiler Project at Rice have contributed to this work in many ways. Tim Harvey acted as a sounding board for many of the ideas presented here as well as a diligent proofreader of this text. Cliff Click provided the example shown in Figure 5. Vivek Sarkar of IBM has provided support for Taylor Simpson; Reid Tatge of TI has provided support for both Chris Vick and Keith Cooper; the Advanced Research Projects Agency provides the major funding for the Massively Scalar Compiler Project. All of these people deserve our sincere thanks.

References

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

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

[3] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–11, San Diego, California, January 1988.

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

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

[6] 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.

[7] 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.

[8] 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.*

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

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

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

[12] 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.

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

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

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.*

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

[21] 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.

[22] 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.

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

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

[25] 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.

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

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

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

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

[30] 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.

[31] 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.*

[32] 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.*