Strength Reduction via SSAPRE

Robert Kennedy, Fred Chow, Peter Dahl,
Shin-Ming Liu, Raymond Lo and Mark Streich

Silicon Graphics Computer Systems
2011 N. Shoreline Blvd.
Mountain View, CA 94043

Contact: Robert Kennedy (E-mail: rkennedy@mti.sgi.com, Tel.: USA 650-933-3336)

Abstract. We present techniques that allow strength reduction to be performed concurrently with partial redundancy elimination in the SSAPRE framework. By sharing the characteristics inherent to SSAPRE, the resulting strength reduction algorithm exhibits many interesting attributes. We compare various aspects of the new strength reduction algorithm with previous strength reduction algorithms. We also outline and discuss our implementation of the closely related linear function test replacement optimization under the same framework.

Keywords. Code motion, data flow analysis, dead code elimination, induction variables, linear function test replacement, partial redundancy elimination, program optimization, program analysis, program transformation, static single assignment, strength reduction.

1 Introduction

Strength reduction refers to program optimization techniques in which expensive or slow operations are replaced by more efficient and faster ones. Today, the term is used mostly to refer to the common case in which computations involving multiplications and additions are transformed to use only additions. This transformation is of universal importance because it benefits any DO loop with arrays indexed by induction variables.

There are mainly two families of methods to perform strength reduction in an optimizing compiler. The first family treats strength reduction as a loop optimization problem that requires explicit detection of induction variables, and implements the technique as a stand-alone algorithm [CK77, ACK81, CP91, CSV95]. The second family effects strength reduction by data flow analysis in the context of partial redundancy elimination [JD82, Cho83, Dha89, KRS93]. The method described in this paper falls in the second family.

Partial redundancy elimination (PRE) is a powerful optimization technique first developed by Morel and Renvoise [MR79]. The technique removes partial redundancies in the program through data flow analysis. Since global common subexpressions and loop-invariant computations are special cases of partial redundancies, PRE handles them elegantly. Joshi and Dhamdhere [JD82, Dha89] and Chow [Cho83] independently describe techniques that allow a PRE implementation to simultaneously perform strength reduction. In this framework, strength reduction does not depend on identifying loop induction variables, and is not restricted to loops. In [KRS92, KRS94], Knoop *et al.* give an alternative PRE algorithm called lazy code motion that improves on Morel and Renvoise's results by avoiding unnecessary code movements, and by removing the bidirectional nature of the original PRE data flow equations. They also presented the lazy strength reduction algorithm [KRS93] that combines strength reduction with lazy code motion.

A new algorithm to perform PRE, called SSAPRE, was recently developed by our team at Silicon Graphics [CCK+97]. The development of this new algorithm was motivated by the fact that traditional data flow analysis based on bit vectors does not interface well with the static single assignment (SSA) form of program representation. In contrast, the SSAPRE algorithm takes advantage of the SSA representation and intrinsically produces its optimized output in the same SSA form. It does not use bit vectors but works instead on one expression at a time, using the *use-def* edges represented in SSA to propagate data flow information. The SSAPRE algorithm thus exhibits the same attributes of sparseness inherent in other SSA-based optimization algorithms. In this paper, we present extensions that enable SSAPRE to perform strength reduction.

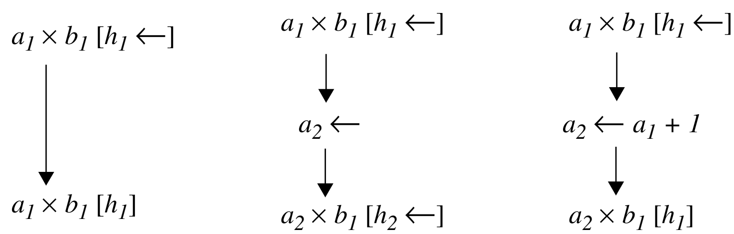
The rest of this paper is organized as follows. We first give an overview of SSAPRE and the basis for extending it to cover strength reduction. Then, in Section 3, we present the SSAPRE-based strength reduction algorithm as a collection of extensions to SSAPRE. In Section 4, we compare SSAPRE-based strength reduction with prior approaches. An optimization closely related to strength reduction is linear function test replacement; in Section 5, we outline our implementation of this technique in the SSAPRE-based strength reduction framework, and discuss the outcome. In Section 6, we conclude by summarizing the contributions of this work.

2 Overview of Approach

Most of the analysis work of PRE is in determining where in the program to insert computations. These insertions cause partially redundant computations to become fully redundant. The partial redundancy problem is thus converted to the full redundancy problem, which can be easily solved.

SSAPRE performs PRE one expression at a time, so it suffices to describe the algorithm with respect to a given expression, say, $a \times b$. SSAPRE consists of six separate steps. The first two steps, Φ-*Insertion* and *Rename,* construct an SSA form for the hypothetical temporary h that represents the value of the expression. The next two steps, *DownSafety* and *WillBeAvail*, perform sparse computation of global data flow attributes based on the SSA graph for h. The fifth step, *Finalize*, determines where in the program to insert computations of the expression, and marks those computations that need to be saved and those that are redundant, and determines the use-def relationship among SSA versions of the real expression temporary t. The last step, *CodeMotion,* transforms the code to form the optimized program.

The *Rename* step plays the important role of identifying redundant computations of the expression. In SSAPRE without strength reduction, two occurrences of $a \times b$ that are assigned identical h-versions compute identical values, as illustrated in Fig. 1(a). But if



(a) redundant (b) not redundant (c) redundant if repaired

Fig. 1. Types of occurrence relationships

a is assigned after the first occurrence, the second occurrence is not redundant since it computes a different value. In Fig. 1(b). we call the assignment to *a* a *killing definition,* since it kills the value computed by the first occurrence of $a \times b$. Now, if we recognize that $a \times b$ is a strength reduction candidate, and if the assignment to *a* is an incremental update, it merely *injures* the first computation. The injured computation can be *repaired* by incrementally updating the expression temporary *t* by a corresponding amount, so that the repaired value can be used at the second occurrence instead of recomputing it. Fig. 1(c) illustrates this third situation. We call the incremental update to *a* an *injuring definition* or simply an *injury.* Since injured computations can be repaired, injuries do not justify assigning new *h*-versions. By assigning h_1 to both $a_1 \times b_1$ and $a_2 \times b_1$, we expose the redundancy between these two occurrences subject to the injury being repaired.

Because we regard injuring definitions as being transparent, the SSA graph built for $a \times b$ by the Φ-*Insertion* and *Rename* steps can exhibit more connectivity and more opportunities for redundancy elimination than if strength reduction is not performed. Thus, the Φ-*Insertion* and *Rename* steps need to be enhanced to ignore injuries in constructing the SSA graph. The generation of injury-repairing code is performed in the *CodeMotion* step. The changes to support strength reduction in the SSAPRE algorithm are confined to the Φ-*Insertion*, *Rename* and *CodeMotion* steps.

3 Strength Reduction Algorithm

We now give more detailed description of the extensions to incorporate strength reduction in the SSAPRE algorithm. The reader is referred to [CCK+97] for a full discussion of the foundation of SSAPRE, and for details of the parts of the algorithm unrelated to strength reduction.

3.1 Strength Reduction Candidates

The set of expressions we regard as strength reduction candidates is determined by the set of assignment statements that we view as injuring definitions. The current implementation handles injuring definitions of the form:

$$a \leftarrow a \pm \langle \text{expr} \rangle$$

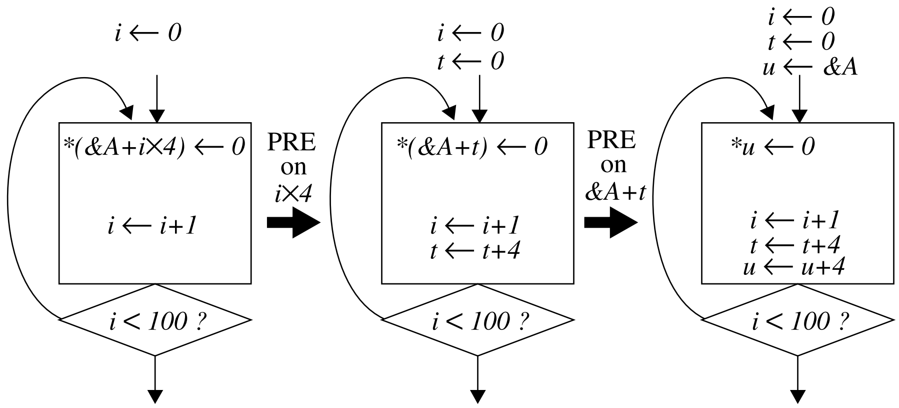


Fig. 2. Strength reduction of nested candidates

Consequently, we handle only strength reduction candidates of the forms $a \pm b$, $-a$ and $a \times b$. For $a \times b$, there is the additional restriction that one of the following must apply:

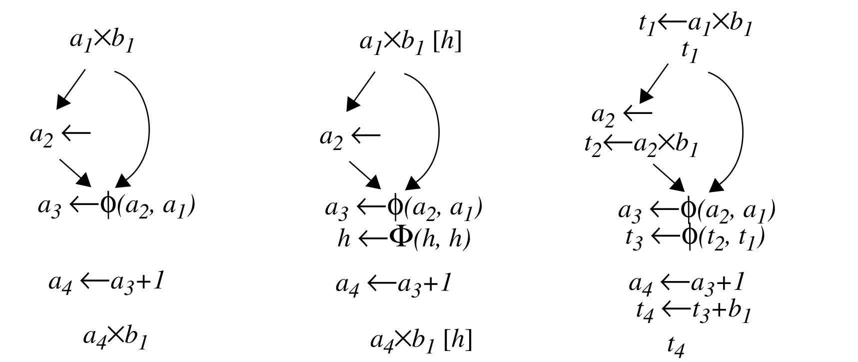
- ⟨expr⟩ is 1, or
- both ⟨expr⟩ and *b* are constants, or
- the injury is inside a loop, ⟨expr⟩ is loop-invariant, and b is either loop-invariant or is another induction variable with respect to the loop.

This restriction is needed to ensure that the code inserted to repair the injury will not contain any multiplication operation after constant folding and code motion; such a multiplication would defeat the benefit of strength reduction. We do not depend on *a* being a loop induction variable.

Our definition of strength reduction candidates only covers *non-compound* expressions, which are expressions consisting of a single operator with only leaf operands. This is consistent with the overall SSAPRE approach that relies on a subtree being converted to a temporary before the parent operator is processed. If the subtree remains intact, no PRE opportunity exists for the parent operator, and hence no strength reduction is possible. This bottom-up approach allows us to automatically strength-reduce large expression trees made up of any combination of the $\pm$, $-$ (negate) and $\times$ operators. This is illustrated by Fig. 2, which shows a loop that initializes an array *A* of 100 elements to 0. The induction expression of the loop is *&A+i×4.* Each application of strength reduction creates a new induction variable. We depend on a later dead code elimination phase [CFR+91] to eliminate any extra induction variables created

3.2 Φ-*Insertion* Step

One purpose of inserting Φ's for the hypothetical temporary *h* is to capture all possible insertion points for the current expression under PRE. Inserting too few Φ's will cause some PRE opportunities to be missed, among other problems. On the other hand, inserting more Φ's than needed will have a negative impact on the efficiency of the algo-



(a) original program (b) after Φ insertion, before renaming (c) after strength reduction

Fig. 3. Enhanced Φ insertion allows more strength reduction

rithm, because a larger SSA graph will be constructed.

As described in [CCK+97], Φ's are inserted according to two criteria. First, Φ's are inserted at the iterated dominance frontiers (DF^+) of each occurrence of the expression. These Φ's are not affected by strength reduction. Second, a Φ can be inserted where there is a ϕ for a variable[1] contained in the expression, because it indicates an alteration of the expression reaches that merge point. The SSAPRE algorithm performs this second type of Φ insertion in a *demand-driven* way; a Φ is inserted only if the expression is partially anticipated. When we recognize that a definition is an injury under strength reduction, an expression not otherwise partially anticipated can be partially anticipated, so some Φ's omitted in the absence of strength reduction must be included. Therefore, the demand-driven Φ insertion algorithm needs to be enhanced to enable strength reduction. Fig. 3 gives an example of this situation. When we check for definition by ϕ's in the algorithm, we continue up the use-def edge at each injury to look for definitions by ϕ's instead of stopping at the injury. From the use of a_4 in Fig. 3(a), we arrive at its definition through the incremental update in $a_4 \leftarrow a_3+1$. Recognizing that this is an injuring definition, we continue upward along the use-def edge from a_3 and arrive at its definition by ϕ, at which point we insert a Φ for *h* in Fig. 3(b). At the end of strength reduction, this Φ for *h* will be replaced by a ϕ for the real temporary *t,* as shown in Fig. 3(c).

Fig. 4 gives the extended version of the Φ–*Insertion* step that handles strength reduction. The parts that differ from the original algorithm[2] are highlighted in bold.

3.3 *Rename* Step

In the previous subsection, we show how the Φ–*Insertion* step inserts more Φ's in the presence of strength reduction, in effect creating more opportunities for code motion

---

1. Following the convention in [CCK+97], we use ϕ in the SSA form for variables and Φ in the SSA form for expressions.

2. The original algorithm is given in Fig. 4 of [CCK+97].

```
procedure Φ-Insertion
   for each expression E_i do {
      DF_phis[i] ⟵ {}
      for each variable j in E_i do
         Var_phis[i][j] ⟵ {}
   }
   for each occurrence X of E_i in program do {
      DF_phis[i] ⟵ DF_phis[i] ∪ DF+(X)
      for each variable occurrence v in X do {
         while (v is defined by injuring-def) do
            v ⟵ previous version in r.h.s. of injuring-def
         if (v is defined by ϕ) {
            j ⟵ index of v in X
            Set_var_phis(Phi(v), i, j)
         }
      }
   }
   for each expression E_i do
      for each variable j in E_i do
         DF_phis[i] ⟵ DF_phis[i] ∪ Var_phis[i][j]
      insert Φ's for E_i according to DF_phis[i]
   }
end Φ-Insertion

procedure Set_var_phis(phi, i, j)
   if (phi ∉ Var_phis[i][j]) {
      Var_phis[i][j] ⟵ Var_phis[i][j] ∪ {phi}
      for each operand v in phi do {
         while (v is defined by injuring-def) do
            v ⟵ previous version in r.h.s. of injuring-def
         if (v is defined by ϕ)
            Set_var_phis(Phi(v), i, j)
      }
   }
end Set_var_phis
```

Fig. 4. Algorithm for enhanced Φ *insertion*

than if strength reduction is not performed. In contrast, the *Rename* step assigns more occurrences of the expression to the same *h*-version, thus enabling more redundancies to be identified than if strength reduction is not performed. Again, the enhancement to the *Rename* step is in dealing with injuring definitions.

In the basic form of the algorithm,[1] *Rename* keeps track of the current version of the expression and the variables contained in it by maintaining rename stacks for each of them while conducting a preorder traversal of the dominator tree of the program.[2] There are three kinds of occurrences of the expression: (1) the expressions in the original pro-

1. For the sake of simplicity, we present the enhancement with respect to the basic *Rename* algorithm. Similar enhancement can be applied to the more practical *Delayed Renaming* algorithm presented in Section 6.2 of [CCK+97].

2. The use of rename stacks during preorder traversal of the dominator tree originated in [CFR+91].

gram, which we call *real* occurrences; (2) the Φ's inserted by Φ–*Insertion*; and (3) Φ operands, which are regarded as occurring at the exits of the predecessor nodes of the corresponding edges. Upon encountering an occurrence q of the expression *E,* if q is a Φ occurrence, we assign q a new *h*-version. If q is a real occurrence or a Φ operand, we check the current version of every variable in the expression *E* (*i.e.,* the version at the top of each variable's renaming stack) against the version of the corresponding variable in the occurrence on the top of *E*'s rename stack. If all the variable versions match, the occurrence as given by the top of *E*'s rename stack is still intact and we assign q the same *h*-version. If any variable version does not match, we check if q is an injured form of the occurrence at the top of *E*'s rename stack, as follows:

> Let p be the occurrence at the top of *E's* rename stack. For each variable v in *E*, we want to determine if its version v_q in q is an injury-updated version of v_p in p. Because we maintain the rename stack in a preorder traversal of the dominator tree,[1] p must dominate the current occurrence q. Starting at v_q, we look up v_q's definition to determine whether v_q is an injured form of v_p's value. If v_q is not defined by an injuring definition, we conclude that q is not an injured occurrence, and must be given a new *h*-version. If v_q is defined by an injury, we look up v_r in the right hand side of the injuring definition and if v_r and v_p are the same version of v, we conclude that v_q is an injured form of v_p's value; otherwise we recursively determine whether v_r is an injured form of v_p's value. If v_q is an injury-updated version of v_p, eventually we reach v_p. If v_p or any v_r is defined by a killing definition or a ϕ, we stop and conclude that q is not an injured occurrence.

If q is an injured occurrence, we assign q the same *h*-version as given by the top of *E*'s rename stack. If q is not an injured occurrence, we follow the same procedure as when we were not performing strength reduction: (a) if q is a real occurrence, we assign q a new version; (b) if q is a Φ operand, we assign the special version $\perp$ to that Φ operand to denote that no evaluation of *E* reaches that point. Finally, we push q on *E*'s stack and proceed.

3.4 *CodeMotion* Step

The three intermediate steps in SSAPRE: *DownSafety, WillBeAvail* and *Finalize,* do not require enhancement to perform strength reduction. These steps apply data flow analyses to the SSA graph constructed by *Rename*, and arrive at the final SSA graph for the real temporary *t.* In this final SSA graph, occurrences of the expression are marked by the *save* and *reload* flags, and places to insert computations of the expression are shown. The *CodeMotion* step introduces assignments to and uses of the real temporary *t,* and transforms the code according to the SSA graph. Under strength reduction, this

---

1. In the preorder traversal, when we come across a definition while descending the dominator tree, we push the new version on top of the rename stack; when the preorder dominator tree traversal backtracks through the blocks containing the definitions, we pop the corresponding versions off the stack.

step is responsible for generating the injury repair code, which updates *t* to maintain its correct value across injuries to the expression's value.

We want to generate an update to *t* only at places where the injured occurrence is partially anticipated; other updates to *t* would be useless. At the same time, we want to collapse multiple updates to *t* into as few update statements as possible. We now present an algorithm for generating injury repairs that is aimed towards these goals.

Processing of injury repairs occurs in the *CodeMotion* step when we encounter an occurrence of the expression that is marked *reload* or is an operand of a Φ marked *will_be_avail* (both of these situations represent uses of *t*). Suppose *q* is such an occurrence. The definition of *q*'s *h*-version, occurrence *p*,[1] must dominate *q*. All the injuries relevant to *q* can be visited by traversing the use-def edges of each variable in *E* starting at *q* until we reach *p*. Our strategy is to generate the repair at the point of the latest injury before each use of *t*. We define a *need_repair* flag for each injuring definition, initialized to *false*. If *need_repair* is *true*, it means that an update to *t* needs to be generated at the point of injury. We add a pre-pass to the *CodeMotion* step for setting this flag. In the pre-pass, starting from *q*, we reach the first injuring definition and set its *need_repair* flag to *true.* At the end of the pre-pass, those injuries closest to at least one use of *t* will have their *need_repair* flags set to *true.* An injuring definition whose *need_repair* flag is *false* represents an intermediate update to the expression value that can be accounted for in the other update statements.

Injury repairs are generated in the main pass of *CodeMotion*. Starting from an injured occurrence *q*, we arrive at the first injury, which must be marked *need_repair.* There, we insert an update statement for *t* if it has not already been generated. The increment amount for *t* is determined by accumulating, while continuing up the use-def edges, the increment amount from each injuring definition up to but excluding the next injury that is marked *need_repair.* The accumulation also stops when we get to *p*. Fig. 5 gives routines that implement the injury repairing algorithm, and their invocations in the *CodeMotion* step of SSAPRE.

Fig. 6 gives an example that shows the effects of the algorithm. The loop contains branches and four induction variable increments. On strength-reducing *i*×*4*, updates to *t* need to be generated. Our algorithm successfully collapses the third and fourth injury updates into a single one at the bottom of the loop.

Based on our definition of strength reduction candidates in Section 3.1, if the increment amount cannot be folded to a leaf node, it must be either a loop-invariant expression or another strength reduction candidate. We add the expression to SSAPRE's worklist so that it will be optimized when SSAPRE works on it in due course.

There are limitations to what our algorithm can accomplish. In particular, our algorithm does not collapse injury repairs across Φ's.

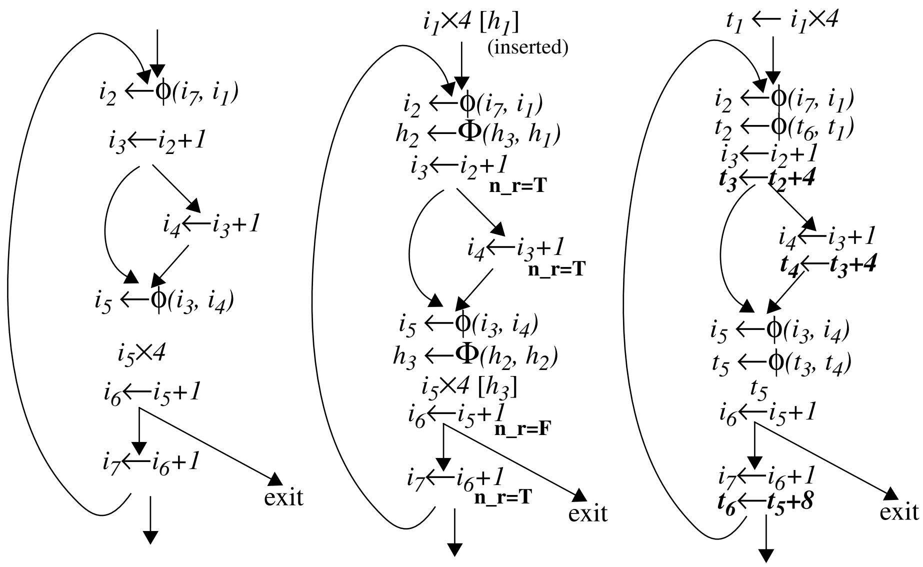
1. At this stage, the definition could also be an inserted occurrence.

```
procedure Set_need_repair(q)
   p ⟵ Avail_def[version(q)]
   for each variable v in E do {
      v_p ⟵ occurrence of v in p
      v_q ⟵ occurrence of v in q
      if (v_p != v_q) {
         D ⟵ defining statement of v_q
         need_repair(D) ⟵ true
      }
   }
end Set_need_repair

procedure Repair_injuries(q)
   p ⟵ Avail_def[version(q)]
   for each variable v in E do {
      v_p ⟵ occurrence of v in p
      v_q ⟵ occurrence of v in q
      if (v_p != v_q) {
         D ⟵ defining statement of v_q
         if (injury-repair for E not yet generated at D) {
            done ⟵ false
            incr_amt ⟵ 0
            S ⟵ D
            do {
               incr_amt ⟵ incr_amt + increment amount for t due to S
               v_r ⟵ occurrence of v in Rhs(S)
               if (v_p == v_r) {
                  done ⟵ true
               } else {
                  S ⟵ defining statement of v_r
                  done ⟵ need_repair(S)
               }
            } while (not done)
            generate t ⟵ t + incr_amt at D
         }
      }
   }
end Repair_injuries

procedure CodeMotion(E)
   initialize need_repair to false for all injuring def statements
   for each occurrence q in E's SSA graph
      if (q is marked reload or q is a Φ operand)
         Set_need_repair(q)
   for each occurrence q in E's SSA graph {
      . . .
      if (q is marked reload or q is a Φ operand)
         Repair_injuries(q)
   }
end CodeMotion
```

Fig. 5. Injury repair algorithm in *CodeMotion* step



(a) original program (b) setting of *need_repair* (**n_r**) (c) final output

Fig. 6. Generating injury repairs

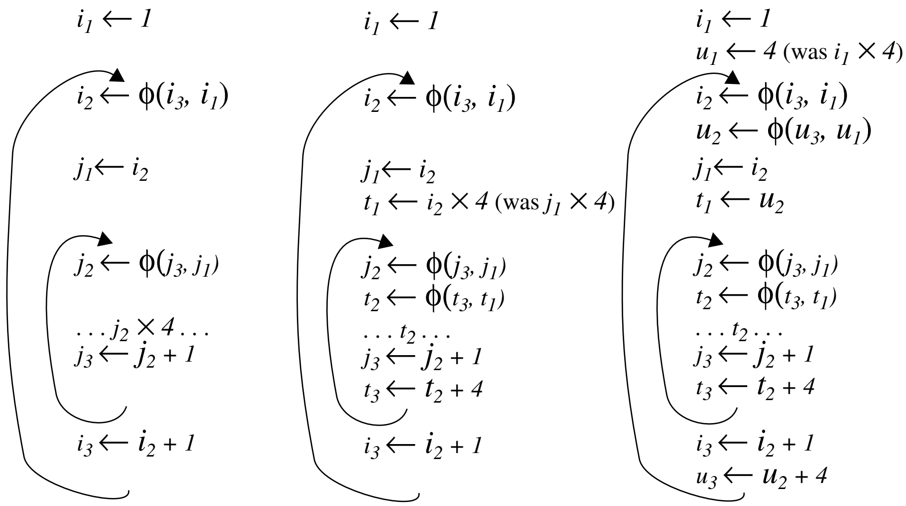
3.5 Second-Order Optimizations

Some opportunities for strength reduction do not appear until other expressions have been strength-reduced. To expose such opportunities, we perform simple copy propagation and constant folding for the expressions inserted by the *CodeMotion* step before moving on to the next PRE candidate. This incremental update after processing each expression allows SSAPRE to handle situations that PRE-based schemes using bit vectors cannot fully optimize without completely repeating PRE. An example appears in Fig. 7, which shows code typical of a loop nest that iterates through the above-diagonal elements of a matrix. The strength reduction candidate $i \times 4$ is not exposed until $j \times 4$ has been strength-reduced.

Fig. 8 shows a strength reduction candidate $i \times j$ that is the product of two induction variables. Our scheme automatically strength-reduces it via the steps shown. Further application of SSAPRE will move the loop-invariant $a \times b$ out of the loop.

4 Comparison with Prior Approaches

As we have remarked in the beginning of this paper, prior methods for performing strength reduction can be divided into two families. The first family is loop-based, and requires stand-alone implementation. The second family is based on data flow analysis, and its implementation is typically integrated with PRE. Although our method belongs to the second family, it combines desirable aspects of both approaches. We begin by comparing our method with other methods belonging to the second family. We follow with a discussion showing how our method incorporates advantages exhibited by the first family.



(a) original program

(b) after PRE on $j \times 4$ and substituting i_2 for j_1

(c) after PRE on $i \times 4$ and substituting 1 for i_1

Fig. 7. Second-order optimization example

Some PRE-based techniques derive most of their advantages from simultaneous execution with PRE. Since PRE is an indispensable optimization in modern optimizing compilers, the resulting strength reduction incurs little or no compile-time overhead. This approach also means lower implementation costs because the amount of code written on top of PRE is small. It does not require control flow analysis, so strength reduction applies also to straight-line code and non-induction variables. Because it relies on the PRE framework, the approach can guarantee optimal placement of the strength reduction candidates. But there are drawbacks in the earlier methods. Without extra work to determine the region constants in a loop, they can apply strength reduction only to constant multipliers and increments by fixed constants. Because they use bit-vector-based data flow analysis, these techniques have to work on all strength reduction candidates at the same time so they cannot easily deal with strength reduction candidates exposed by optimizing other expressions. Special efforts are needed to deal with nested strength reduction candidates [Cho83]. It is nontrivial to determine when not to

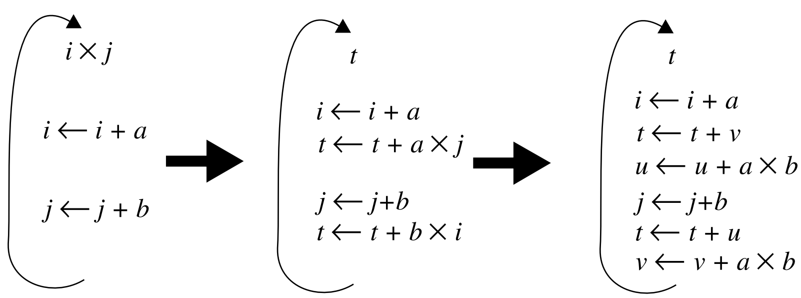


Fig. 8. Strength reduction of $i \times j$

strength-reduce because the number of increments inserted more than offsets the benefit of the saved multiplication [JD82].

In our case, earlier compilation phases have gathered information about the loop structure of the program. Using the SSA representation, we can easily determine the variables and expressions that are loop-invariant, which allows us to form a larger candidate set for strength reduction. Because we work on one candidate at a time, we iteratively handle new strength reduction candidates and loop-invariant expressions formed by earlier strength reduction as discussed in section 3.5.

Working on one candidate at a time could slow our algorithm down because we give up the natural parallelism available through bit vectors. But while data flow analysis based on bit vectors is performed with respect to the control flow graph, our work is based on a sparse graph representation of the strength reduction candidate, so it takes fewer steps to process each candidate. Because our algorithm continuously updates the program representation, even though our base algorithm only applies to expressions consisting of only one operator, we automatically handle large expression trees bottom-up by repeatedly converting each operator node to a temporary.

In [KRS93], Knoop *et al.* describes an algorithm that accumulates multiple incremental updates of t into a smaller number of updates. Their algorithm only works within extended basic blocks. The algorithm we presented in Section 3.4 has greater coverage because it accumulates incremental updates within regions separated by Φ's, which are at least as large as, and often larger than extended basic blocks.

The first family of strength reduction techniques is represented by the classic algorithm by Allen, Cocke and Kennedy, presented in [ACK81] and [CK77]. Researchers at Rice University improved on this algorithm by applying techniques that take advantage of an SSA representation of the program. The method presented in this paper shares the characteristics with these algorithms in processing one expression at a time, though we operate on the entire program instead of a single loop. Processing one expression at a time allows us to easily customize the handling of individual types of expressions and situations. For example, it is possible to extend our algorithm to the other forms of strength reduction candidates as listed in [ACK81]. Because our algorithm is effective in combining multiple incremental updates into fewer updates, cases where the number of increments inserted more than offset the benefit of the saved multiplication are rare.

One drawback of our method is that we cannot easily handle mutually defined induction variables, of the form:

```
while (. .) {
    i = j + 3;
    . . . .
    j = i + 2;
}
```

We instead rely on an earlier phase in the compiler that transforms the loop to use a single well-formed induction variable, so that the above form will never appear as input to our strength reduction phase. The techniques to do this transformation are described in [LLC96].

5 Linear Function Test Replacement

Strength reduction paves the way for another optimization called linear function test replacement (LFTR) [CK77]. In the example of Fig. 2, if we convert the loop termination test $i < 100$ to the equivalent $u < (\&A+400)$, i can be removed from the loop by subsequent dead code elimination. LFTR involves the following two tasks:

- locating each comparison operator such that replacing the test will allow the original induction variable to be removed by dead code elimination;
- finding the best strength reduction candidate to replace the induction variable with.

Because the replacing candidate must be a temporary formed from strength reduction, LFTR depends on the results of strength reduction. In the loop-based family of strength reduction techniques, LFTR is not difficult since any required analysis is restricted to the loop in question. In the family of strength reduction techniques based on data flow analysis, an LFTR algorithm was described by Chow [Cho83]. Chow's PRE-based LFTR algorithm was implemented as a post-pass to PRE. It analyzes the global data flow attributes computed by PRE at the basic block containing the comparison operator and selects the largest strength reduction candidate expression that is available at the comparison operator to do the replacement. The PRE-based LFTR algorithm does not require identification of induction variables, and is not restricted to loops. After dead code elimination, a final pass over the test replacement candidates suppresses those replacements whose induction variables were not eliminated.

Under SSAPRE-based strength reduction, the data flow information for each expression is stored in data structures that we reclaim for use in processing the next expression. This makes it impossible to perform LFTR as a post-pass, because no data flow information is available then. Instead, LFTR needs to be performed concurrently with strength reduction and SSAPRE:

When we work on an expression under SSAPRE, we use a new kind of occurrence node, each instance of which represents an LFTR candidate. We call these nodes *comparison* occurrences. They are of the form:

$$x \; op \; \langle \text{expr} \rangle \qquad \text{where } op \text{ is one of } <, >, <=, >=, ==, !=$$

For example, for the strength reduction candidate $i \times j$, we recognize $i < \langle\text{expr}\rangle$ and $j < \langle\text{expr}\rangle$ as comparison occurrences. These comparison occurrences are assigned *h*-versions like other occurrences in the *Rename* step. If a comparison occurrence is not assigned a new *h*-version, it implies the strength reduction candidate that can be used for test replacement is potentially available at the comparison operation. The test replacement is performed in the *CodeMotion* step, after verifying that the strength reduction candidate is actually available at the comparison. For the strength reduction candidate

$i \times j$, $i < \langle \text{expr} \rangle$ will be replaced by $t < \langle \text{expr} \rangle \times j$, and $j < \langle \text{expr} \rangle$ will be replaced by $t < i \times \langle \text{expr} \rangle$.

As with strength reduction, we also impose restrictions on the forms of the comparison expressions and strength reduction candidates handled to ensure that the new forms of the comparison expressions are at least as efficient as the original ones. For straight-line code, we require that the new right hand side be reducible to a leaf node via constant folding. For loops, we require that the new right hand side be either loop-invariant or be another strength reduction candidate, so that it will eventually be converted to a leaf node after being processed by SSAPRE.

By performing LFTR concurrently with strength reduction and SSAPRE, we avoid a separate LFTR phase in the optimizer. Instead, we incur some incremental overhead in the PRE phase to do the LFTR work. Our implementation of SSAPRE handles only non-compound expressions. Though this simplifies the implementation, it necessitates multiple test replacements to achieve the effect of replacing by a compound expression. For example, the program fragment in Fig. 2 requires two separate LFTR's: one to transform the loop termination test from $i < 100$ to $t < 400$, and another to further change it to $u < (\&A+400)$.

Because we perform data flow analysis on one expression at a time, gathering all the possible replacing candidates together and picking the best from them is out of the question. As a result, we have to perform LFTR at the first opportunity. Depending on the order of the expressions in SSAPRE's worklist, we may not end up choosing the best candidate. For the loop termination test $i < 100$, using the induction expression $i \times 4$ for test replacement is better than using $i \times j$, because using $i \times j$ will cause $100 \times j$ to be inserted at the loop header. We can mitigate this problem by ordering SSAPRE's worklist such that expressions with constant operands are processed earlier.

Since we incrementally update the program representation in LFTR, after dead code elimination we cannot easily undo test replacements that did not allow any induction variable to be eliminated. Useless LFTR's result in unnecessary loop-invariant computations being inserted at the loop header. Their negative impact on performance is noticeable only in loops with small trip count. One possible solution to this problem is to perform a preliminary induction variable analysis pass to tag induction variables that are either live at loop exit or have uses not associated with induction expressions inside the loop. We can then avoid performing LFTR on such variables, since they cannot be removed by dead code elimination even after strength reduction and LFTR.

6 Conclusion

We have presented techniques that allow strength reduction to be integrated into the SSAPRE framework. The techniques are easy to implement, and they incur little compile-time overhead. From a different angle, they bring forth greater opportunities for code motion, making SSAPRE more powerful. The resulting strength reduction inherits all the characteristics of SSAPRE: output maintained in SSA form, sparse representation, sparse computation of global data flow attributes, and ease of handling large expression trees. Because it works on one expression at a time, it also combines the best of the two families of strength reduction algorithms.

In addition to strength reduction, we have also integrated linear function test replacement into the SSAPRE framework. LFTR presents more integration problems than strength reduction does, but with careful engineering, those problems are surmountable in practice.

References

[ACK81] Allen, F., Cocke, J. and Kennedy, K. Reduction of Operator Strength, in Muchnick, S. and Jones, N. (editors) *Program Flow Analysis: Theory and Applications,* Prentice Hall, 1981, pp. 79-101.

[CP91] Cai, J. and Paige, R. Look Ma, No Hashing, and No Arrays Neither. Proceedings of the 18th Symposium on Principles of Programming Languages, January 1991, pp. 143-154.

[CCK+97] Chow, F., Chan, S., Kennedy, R., Liu, S., Lo, R. and Tu, P. A New Algorithm for Partial Redundancy Elimination based on SSA Form. Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation, June 1997, pp. 273-286.

[Cho83] Chow, F. "A Portable Machine-independent Global Optimizer — Design and Measurements," Ph.D. Thesis and Technical Report 83-254, Computer System Lab, Stanford University, Dec. 1983.

[CK77] Cocke, J. and Kennedy, K. An Algorithm for Reduction of Operator Strength. *Communications of the ACM,* 20:11 (1977) pp. 850-856.

[CFR+91] Cytron, R., Ferrante, J., Rosen B., Wegman, M. and Zadeck, K., Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems,* October 1991, pp. 451-490.

[CSV95] Cooper, K., Simpson, T. and Vick, C. Operator Strength Reduction, Technical Report CRPC-TR95635-S, Rice University, October 1995.

[Dha89] Dhamdhere, D. A New Algorithm for Composite Hoisting and Strength Reduction Optimization (+ Corrigendum). *International Journal of Computer Mathematics 27* (1989), pp. 1-14 (+ 31-32).

[JD82] Joshi, S. and Dhamdhere, D. A Composite Hoisting-Strength Reduction Transformation for Global Program Optimization, Part I and II. *International Journal of Computer Mathematics,* II (1982), pp. 21-41, 111-126.

[KRS92] Knoop, J., Ruthing, O. and Steffen, B., Lazy Code Motion. Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation, June 1992, pp. 224-234.

[KRS93] Knoop, J., Ruthing, O. and Steffen, B., Lazy Strength Reduction. *Journal of Programming Languages* (1993) 1, pp. 71-91.

[KRS94] Knoop, J., Ruthing, O. and Steffen, B., Optimal Code Motion: Theory and Practice. *ACM Transactions on Programming Languages and Systems,* October 1994, pp. 1117-1155.

[LLC96] Liu, S., Lo, R. and Chow, F.. Loop Induction Variable Canonicalization in Parallelizing Compilers. Proceedings of the Fourth International Conference on Parallel Architectures and Compilation Techniques, October 1996, pp. 228-237.

[MR79] Morel, E. and Renvoise, C. Global Optimization by Suppression of Partial Redundancies. *Communications of the ACM 22(2)*, February 1979, pp. 96-103.