Partial Redundancy Elimination in SSA Form

ROBERT KENNEDY, SUN CHAN, SHIN-MING LIU, RAYMOND LO, PENG TU[1]
and
FRED CHOW
Silicon Graphics Computer Systems

The SSAPRE algorithm for performing partial redundancy elimination based entirely on SSA form is presented. The algorithm is formulated based on a new conceptual framework, the factored redundancy graph, for analyzing redundancy, and represents the first sparse approach to the classical problem of partial redundancy elimination. At the same time, it provides new perspectives on the problem and on methods for its solution. With the algorithm description, theorems and their proofs are given showing that the algorithm produces the best possible code by the criteria of computational optimality and lifetime optimality of the introduced temporaries. In addition to the base algorithm, a practical implementation of SSAPRE that exhibits additional compile-time efficiencies is described. In closing, measurement statistics are provided that characterize the instances of the partial redundancy problem from a set of benchmark programs and compare optimization time spent by an implementation of SSAPRE against a classical partial redundancy elimination implementation. The data lend insight into the nature of partial redundancy elimination and demonstrate the expediency of this new approach.

Categories and Subject Descriptors: D.3.3 [**Programing Languages**]: Language Constructs—*control structures*; *data types and structures*; *procedures, functions and subroutines*; D.3.4 [**Programming Languages**]: Processors—*compilers*; *optimization*; I.1.2 [**Algebraic Manipulation**]: Algorithms—*analysis of algorithms*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program transformation*

General Terms: Partial Redundancy, Code Motion, Static Single Assignment Form

Additional Key Words and Phrases: common subexpressions, data flow analysis, program optimization

1. INTRODUCTION

Partial redundancy elimination (PRE) is a powerful optimization technique first developed by Morel and Renvoise [Morel and Renvoise 1979]. The technique removes partial redundancies in the program by performing data flow analysis that solves for code placements. Since global common subexpressions and loop-invariant

Authors' address: Silicon Graphics Computer Systems, 2011 N. Shoreline Blvd., Mountain View, CA 94043.

[1]Peng Tu's present address: Intel Corporation, 2200 Mission College Blvd., Santa Clara, CA 95052.

computations are special cases of partial redundancies, they are subsumed by PRE. As a result, PRE has become the most important component in many global optimizers [Chow 1983; Chow et al. 1986; Schwarz et al. 1988; Briggs and Cooper 1994; Simpson 1996]. An alternative placement strategy called lazy code motion [Knoop et al. 1992; Knoop et al. 1994] 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. The result of lazy code motion is optimal: the number of computations cannot be further reduced by safe code motion [Kennedy 1972], and the lifetimes of the temporaries introduced are minimized. [Drechsler and Stadel 1993] gives a variant of the lazy code motion algorithm that is based on a different data flow framework. [Muchnick 1997] gives a good introduction to the problem of partial redundancy elimination and to classical methods for its solution.

Each of the above approaches to PRE is based on a bit-vector formulation of the problem, and on the iterative solution of data flow equations. This paper presents a new approach called SSAPRE [Chow et al. 1997] that shares the optimality properties of the best prior work [Knoop et al. 1992; Knoop et al. 1994; Drechsler and Stadel 1993] and that is based on static single assignment form. Static single assignment form (SSA) is a popular program representation in modern optimizing compilers. Its versatility stems from the fact that, in addition to representing the program, it provides accurate use-definition (use-def) relationships among the program variables in a concise form [Cytron et al. 1991; Wolfe 1996; Chow et al. 1996]. Many efficient global optimization algorithms have been developed based on SSA. Among these optimizations are dead store elimination [Cytron et al. 1991], constant propagation [Wegman and Zadeck 1991], value numbering [Alpern et al. 1988; Rosen et al. 1988; Briggs et al. 1997], induction variable analysis [Gerlek et al. 1995; Liu et al. 1996], live range computation [Gerlek et al. 1994] and global code motion [Click 1995]. Until recently, most uses of SSA have been restricted to solving problems based essentially on program variables. SSA could not readily be applied to solving expression-based problems because the concept of use-def for expressions is less obvious than for variables. This difficulty was mentioned by Dhamdhere *et al.* in the conclusion of [Dhamdhere et al. 1992]. They state, essentially, that there is no clear connection between the use-def information for variables represented by SSA form and the redundancy properties for expressions. By demonstrating such a connection and exploiting it, our work shows that an SSA-based approach to PRE and other expression-based problems is not only plausible, but also enlightening and practical. Although this paper addresses only the PRE problem, other expression-based problems can be addressed based on the framework presented.

There are many reasons why an SSA-based solution to an optimization problem is desirable. Optimizations based on SSA all share the common characteristic that they do not require traditional iterative data flow analysis in their solutions. They all take advantage of the *sparse* representation of SSA. In a sparse form, information associated with an object is represented only at places where it changes, or when the object actually occurs in the program. Sparse representations typically conserve memory space by avoiding needless duplication of data. Information can be propagated through a sparse representation in a smaller number of steps than through a dense structure, speeding up most algorithms. To benefit fully from sparseness, one must often sacrifice the parallelism that can be achieved in many techniques that operate on the entire program at once. For example, traditional data flow analysis based on bit vectors can operate on all program expressions in parallel. Although sparse schemes give up this parallelism, operating on each element separately allows optimization decisions to be customized for each one.

Another advantage of SSA is that it allows a global optimization to efficiently subsume the local version of the same optimization. Depending on the nature of the optimization, traditional frameworks typically require two separate implementations for efficiency's sake: a global version that uses bit vectors in each basic block, and a simpler and faster local version that performs the same optimization within a basic block. Global implementations of an optimization can handle the local version's job, but usually at substantially higher cost. In contrast, SSA-based optimization algorithms do not need to distinguish between global and local optimizations because SSA directly exposes the use-def relationships in the program. The same algorithm can handle both global and local versions of an optimization simultaneously and efficiently. The amount of effort required to implement an optimization can thus be correspondingly reduced. Similar reductions in implementation effort for some optimizations can be had through prior sparse techniques similar to ours [Choi et al. 1991], although that work did not apply its techniques to expression-based optimization problems.

Further motivation for this work comes from the fact that traditional data flow analysis based on bit vectors does not interface well with the SSA form of program representation. The use-def information encoded in SSA has to be converted to bit-vector form in order to apply the bit-vector-based algorithms. This process involves scanning the contents of each basic block in the program to initialize the local data flow attributes in bit-vector form. Experience has shown that this dense initialization of data flow information often takes more time than the solution of the data flow equations. After transformation, the program has to be put back into SSA form if subsequent SSA-based optimizations are desired. Such repeated updates to SSA form due to arbitrary modifications to the program can add up to substantial compile-time overhead [Choi et al. 1996]. In contrast, the SSAPRE algorithm exploits the built-in use-def information in its input SSA form, and intrinsically produces its optimized output in SSA form. It performs data flow propagation based on sparse graphs that it constructs. The entire program is maintained in valid SSA form as SSAPRE iterates through the PRE candidates.

The rest of this paper is organized as follows. Section 2 briefly reviews the fundamentals of the SSA form and presents the factored redundancy graph (FRG) which forms the basis of our sparse approach to PRE. Section 3 describes the SSAPRE algorithm in detail, while stating related lemmas with proofs. Section 4 discusses the theoretical aspects of the SSAPRE algorithm, and verifies its correctness and optimality. Section 5 discusses some practical issues related to an efficient and effective implementation of SSAPRE. Section 6 compares and contrasts the steps in SSAPRE with bit-vector-based PRE, and analyzes the complexity of the SSAPRE algorithm. Section 7 provides measurement data that compare an implementation of our algorithm against a bit-vector PRE implementation and that characterize the partial redundancy problems based on our approach across a set of benchmark programs. Section 8 concludes by discussing the implications of this work.

2. SSA AND SPARSE PRE

As background for the SSAPRE algorithm, we briefly define some terms, review some characteristics of SSA form, and discuss some properties of redundancy among computations in a program. We define the concept of a redundancy relation for a program computation and present the factored redundancy graph (FRG) which we use to represent such a relation. We briefly review the connections between SSA form and the classical use-definition relation for a program variable, and we show that the FRG and the redundancy relation share the same connections. This analogy between SSA and our factored representation of redundancy in the program is the foundation of the SSAPRE algorithm's ability to operate directly on an input program in SSA form, and to produce its output directly in SSA form.

2.1 Control Flow and Dominance

We assume the code for the program being optimized has been partitioned into *basic blocks* with the property that control may enter a basic block only at the beginning and may leave only at the end. The *control flow graph* has the basic blocks as its nodes, and has an edge from block B_1 to block B_2 if and only if control can transfer directly from the end of B_1 to the beginning of B_2. Without loss of generality, we assume the program has a unique *entry* and a unique *exit* block, and that every block in the program lies on some path from entry to exit.

A block B_1 is said to *dominate* the block B_2 if every control flow path from the program entry to B_2 encounters B_1. We say B_1 *strictly dominates* B_2 if B_1 dominates B_2 and $B_1 \neq B_2$. If B_1 strictly dominates B_2 and no block other than B_1 on any control flow path from B_1 to B_2 strictly dominates B_2, we say B_1 is the *immediate dominator* of B_2.

A *dominator tree*, abbreviated DT, is a tree whose nodes are the basic blocks of the program, whose root is the program entry block, and in which the parent of each block is that block's immediate dominator. The *dominance frontier* [Cytron et al. 1991] of a block B, abbreviated $\mathrm{DF}(B)$, is the set of blocks not strictly dominated by B and having at least one predecessor dominated by B. The *iterated dominance frontier* [Cytron et al. 1991] of a block B, abbreviated $\mathrm{DF}^+(B)$, is the smallest set of blocks that contains $\mathrm{DF}(B)$ and is a fixed point under pointwise application of $\mathrm{DF}(\cdot)$.

2.2 SSA Form

In this section we give a brief review of the Static Single Assignment (SSA) form of program representation. For greater detail, the reader is referred to, *e.g.*, [Cytron et al. 1991].

DEFINITION 1. *A program is said to be in* SSA form *if each of its variables is defined exactly once, and each use of a variable is dominated by that variable's definition.*

This definition is strict enough that programs with nontrivial control flow require some special consideration if they are to be put in SSA form. Hence we say that in SSA form, the definition of a variable may be an assignment from a special operator denoted ϕ which is used to capture the effects of control flow. The presence of an assignment $y \leftarrow \phi(x^{(1)}, \ldots, x^{(n)})$ in a basic block B means the following:

—B has exactly n predecessors in the control flow graph (one for each operand of the ϕ),

—$x^{(1)}, \ldots, x^{(n)}$ and y are variables in the program, and

—if control arrives in block B from its j^{th} predecessor, y has the value of $x^{(j)}$ at the beginning of B.

An important convention regarding ϕ operators is that for the purpose of the dominance relation among uses and definitions of variables in the program, operands of ϕ are regarded as occurring at the ends of their corresponding predecessor blocks, while the assignment to the ϕ result occurs at the beginning of the block containing the ϕ.

Given a program and its control flow graph, the program can be put in SSA form by assigning a unique *version* to each definition of a variable and placing ϕ operators defining additional versions in basic blocks that are reached by multiple definitions of the same original variable. After such a transformation, each version is viewed as a variable in its own right. Versions are traditionally denoted by applying subscripts to the name of the original program variable, so SSA versions of variable v will be denoted v_1, v_2, and so forth. [Cytron et al. 1991] gives an efficient algorithm to put a program in SSA form using the minimum number of ϕ assignments.

In the remainder of this section, we discuss the use-definition (use-def) relation and its connection with SSA form. The *use-def relation* is a relation between uses of variables and definitions of (assignments to) variables in the program. In a *use-def graph* representing this relation, there is an edge leading from each use of a variable to every reaching definition for that use. In the following discussion we explain that SSA is a factored form of the use-def graph [Wolfe 1996]. Additional details of this connection between SSA form and the use-def relation are contained in [Cytron et al. 1991].

Toward defining the *factored use-def graph*, we make the simplifying assumption that every definition is a killing definition, and let $\{d_1, \ldots, d_n\}$ be the set of definitions reaching some use u. There are use-def edges from u to each of $d_1, \ldots, d_n$. A basic block B is called a ϕ block (factoring point) for u if:

—the beginning of B dominates u; and

—$\exists i_1, i_2$, and paths P_1, P_2 such that:
  —$i_1 \neq i_2$; and
  —d_{i_1} is contained in the first node of P_1 and d_{i_2} is contained in the first node of P_2; and
  —B is the final node on both P_1 and P_2; and
  —P_1 and P_2 have no node in common except B.

Cytron *et al.* showed that the set of ϕ blocks for the uses of an original program variable is contained in the union of the iterated dominance frontiers of the blocks containing real definitions of the variable [Cytron et al. 1991]. Now given the control flow graph and the use-def relation for a variable v, we define the *factored use-def graph* as follows: The nodes of v's factored use-def graph are the uses and definitions of v's use-def relation plus a ϕ node for v in each basic block that is a ϕ block for some use of v. Each of these ϕ nodes represents both the ϕ assignment itself and the collection of ϕ operands in the ϕ block's predecessor blocks. There is an edge in v's factored use-def graph from the node representing each use (including ϕ operands[2]) to the node representing the immediate dominating definition of v, which may be a ϕ.

It is a straightforward exercise to verify that the factored use-def graph for a program variable is equivalent to minimal SSA form for that variable. The correspondence is the obvious one: each use of a given SSA version of a variable in SSA form corresponds to an edge in the factored use-def graph between the node corresponding to the use and the node corresponding to the unique definition for that SSA version.

It is also well known and easy to check that the original use-def relation can be recovered from the factored use-def graph by taking the transitive closure and discarding those edges in the transitive closure that have a ϕ or ϕ operand as an endpoint.

2.3 Foundation of Sparse PRE

In this section, we define several terms and outline the basis of the framework in which we analyze redundancy. One goal of this section is to show connections between our redundancy framework and SSA form for variables; these connections underlie the intuition behind the SSAPRE algorithm's ability to directly generate its output in SSA form. For convenience in our definitions, we assume every basic block in the control flow graph of the program being compiled is reachable from the entry block, and that the program exit is reachable from every basic block.

DEFINITION 2. *If E_1 and E_2 are occurrences of some computation E and there is a control flow path from E_1 to E_2 containing nothing that may alter the value of E, we say that E_2 is* redundant with respect to *E_1.*

Our sparse approach to PRE relies on a representation that can directly expose partial redundancy; such a representation is derived in the following discussion. Suppose an occurrence E_2 is redundant with respect to E_1. We represent this redundancy by a directed edge from E_2 to E_1.

DEFINITION 3. *Let E_1 be an occurrence of some computation E and p be some point in the program. If there is a control flow path from p to E_1 containing nothing that may alter the value of E and containing no occurrence of E between p and E_1, we say that E_1 is* exposed *with respect to p.*

Now let $\Omega = \{E_1, \ldots, E_n\}$ be the set of occurrences with respect to which an occurrence E_0 is exposed and redundant, and let $A = \{a_1, \ldots, a_m\}$ be the set of alterations of E's value (*e.g.*, assignments to operands of E if E is an expression) with respect to which E_0 is exposed. Let $V = A \cup \Omega$. A block B is called a Φ block (factoring point) for E_0 if:

—the beginning of B dominates E_0; and
—$\exists v_1, v_2 \in V$ and paths P_1 and P_2 such that:
  —$v_1 \neq v_2$; and

[2]Recall that ϕ operands are viewed as occurring at the ends of their corresponding predecessor blocks.

—v_1 is contained in the first node of P_1 and v_2 is contained in the first node of P_2; and
—B is the final node of both P_1 and P_2; and
—P_1 and P_2 have no node in common except B.

To take our next step in formulating our algorithm's framework, we need an additional definition:

DEFINITION 4. *We say a computation is* partially available *at some point p in the program if there is a control flow path leading to p from some real occurrence of the computation and not crossing anything that may alter the value of the computation.*

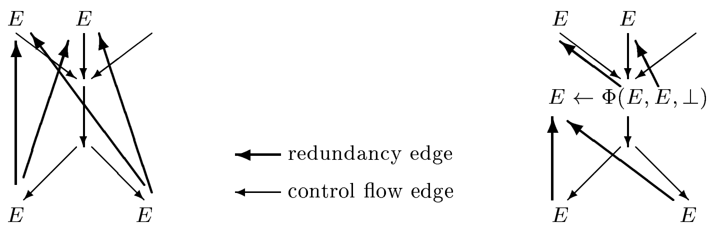
We say an occurrence ω is *partially redundant* if it is an occurrence of a computation that is partially available just before ω.

In the same way that the literature uses a ϕ operator in SSA form to factor the use-def relation for variables, the following paragraph will introduce a Φ operator that factors the redundancy relation for computation occurrences. Just as ϕ operators are viewed as *bona fide* assignments to variables in SSA form, we will regard instances of the factoring operator Φ and operands of these Φ's as computation occurrences in their own right, and we will use the term "real occurrence" to distinguish the occurrences of the computation that correspond to code in the program from Φ and Φ operand occurrences. As in the case of ϕ operators and their operands, we view each Φ as occurring at the beginning of the block in which it appears, and operands of each Φ as occurring at the ends of their corresponding predecessor blocks. There can be operands of Φ that are not partially redundant; these have no counterpart in SSA form, and we denote them by the symbol $\perp$.

Given a partially redundant real or Φ operand occurrence E_0, we define the *representative occurrence* for E_0 as the nearest to E_0 among those Φ or non-partially-redundant real occurrences that dominate E_0. The reader can easily verify that such a representative occurrence is well defined and unique.

Now given the control flow graph and the redundancy relation for a computation E, we define the *factored redundancy graph* (FRG) as follows: The nodes of E's FRG are the real occurrences in E's redundancy relation plus a Φ node for E in each basic block that is a Φ block for some real occurrence of E. There is an *upward edge* in the FRG from each partially redundant real occurrence and each partially redundant Φ operand occurrence to its representative occurrence. Figure 1 shows the upward edges in an example of how the FRG factors the edges of the redundancy relation. The reverse of each upward edge is called a *downward edge*. The set of occurrences made up of a representative occurrence and those occurrences it represents is called a *redundancy class*. The reader may think of redundancy classes as roughly analogous to variable versions in SSA form. Upward edges correspond to use-def edges in SSA (*i.e.*, the edges of the factored use-def graph), and downward edges correspond to def-use edges in SSA. To underscore this analogy, we will sometimes say that the representative occurrence for a redundancy class *defines* the class and its members.

Clearly the FRG, being defined so similarly to the factored use-def graph, has a good deal in common with SSA form. For example, the original redundancy relation can be recovered from the FRG and the control flow graph in much the same way as the full use-def relation is recovered from SSA form. As we will see, the



(a) without factoring (b) factored

Fig. 1. Factoring of redundancy edges

connections are deeper than an abstract analogy between the definitions and their properties: the particular FRG for a computation is connected closely with SSA form for the computation's temporary variable after PRE (See Section 2.5 below).

2.4 Basics of PRE

We say that a computation is *available at a point q in the program along path $\mathcal{P}$* if:

—$\mathcal{P}$ is a path leading to the point q; and

—the computation occurs at some point r on $\mathcal{P}$ with the property that between r and q, $\mathcal{P}$ contains nothing that may alter the value of the computation.

Reiterating Definition 4, a computation is *partially available at q* if there exists a path along which the computation is available at q. A computation is *fully available at q* if it is available at q along every path from the program entry to q.

We say that a computation is *anticipated at a point q along path $\mathcal{P}$* if:

—$\mathcal{P}$ is a path beginning at q; and

—the computation occurs at some point r on $\mathcal{P}$ with the property that between q and r, $\mathcal{P}$ contains nothing that may alter the value of the computation.

A computation is *partially anticipated at q* if there exists a path along which the computation is anticipated at q. A computation is *fully anticipated at q* if it is anticipated at q along every path from q to the program exit. If a computation is fully anticipated at q, we say the point q is *down-safe* with respect to that computation [Knoop et al. 1994].

Following [Knoop et al. 1992], we use the term *placement* to refer to the set of points in the optimized program where a particular computation occurs. We say that a placement is *safe* if optimization has not introduced new values to any path in the program, *i.e.*, if every inserted computation occurs at a point where the computation is fully anticipated or fully available [Kennedy 1972]. This requirement is intended to prevent incorrect behavior of the optimized program in the presence of computations that may cause exceptions (*e.g.*, division by zero). Safety is considered a fundamental requirement in the literature, and all proposed methods for eliminating partial redundancies adhere to this requirement, *e.g.*, [Morel and Renvoise 1979; Chow 1983; Drechsler and Stadel 1988; Dhamdhere 1988; Dhamdhere et al. 1992; Knoop et al. 1992; Drechsler and Stadel 1993]. We say that a placement is *computationally optimal* if no safe placement can result in fewer occurrences of the computation along any path from entry to exit in the program. Computational optimality is an important requirement in partial redundancy elimination,

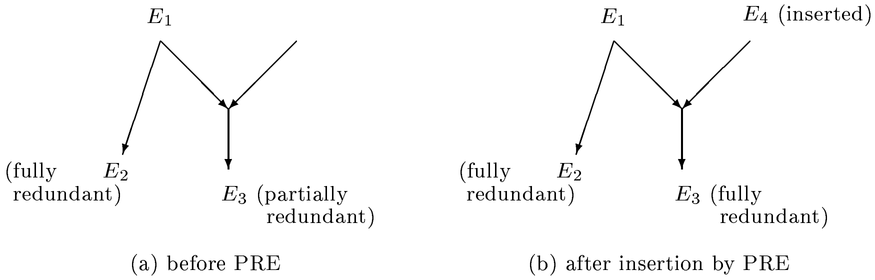


Fig. 2. Full and partial redundancies

but several early methods, *e.g.*, [Morel and Renvoise 1979; Chow 1983], lacked this property. The property was later achieved for these methods by requiring the insertion of synthetic basic blocks along certain control flow edges (see the beginning of Section 3).

Consider the redundancies associated with a computation E that yields a value in the procedure being compiled. An occurrence of E is *fully redundant* if E is fully available just before the point of that occurrence in the program, and fully redundant computations can be safely removed by simply deleting them. E_2 in Figure 2(a) is a fully redundant occurrence. An occurrence of E is *partially redundant* if E is partially available just before the point of the occurrence. E_3 in Figure 2(a) is a partially redundant (but not fully redundant) occurrence. Eliminating strictly partial redundancies involves inserting new computations of E (like E_4 in Figure 2(b)) to render occurrences fully redundant so they can be deleted. Figure 3 shows a larger example program in SSA form before and after optimum PRE has been performed.

When the computations under consideration are expressions that compute values in the program, deleted occurrences are replaced by loads from a temporary variable introduced during optimization. To ensure that this temporary contains the correct value when it is accessed in the optimized program, the value of the expression is saved to the temporary at a subset of the points where the expression is evaluated. An important practical concern is the exacerbation of register pressure that can result from introducing these temporaries. To address this concern, some work in PRE [Chow 1983; Dhamdhere 1988; Drechsler and Stadel 1988] made heuristic modifications to the system of data flow equations introduced in [Morel and Renvoise 1979], but none of these techniques, which are based directly on the work of Morel and Renvoise, achieved the goal of minimizing the lifetimes of the introduced temporaries. In particular, all those PRE schemes would perform code motion that introduced an unnecessary temporary without removing any redundancy in examples like our Figure 9.[3] Lifetime optimality of the introduced temporary variables subject to the constraint of computational optimality was first achieved in [Knoop et al. 1992]. Other research that achieves the same result includes [Drechsler and Stadel 1993] and the present work.

[3]Algorithms based on the framework of Morel and Renvoise make the following harmful transformation on the example of Figure 9: $a + b$ is introduced into blocks 2 and 4 and $a + b$ is deleted from block 6.

2.5 A Central Observation

Suppose that computationally optimum PRE has been performed, replacing expression E with uses of the temporary variable t in places where computations of E were deleted. The central observation leading to our algorithm for PRE is the following:

OBSERVATION 1. *Every edge in the use-def relation for t corresponds directly to a redundancy edge for E, or to a redundancy edge introduced during PRE between a deleted occurrence of E and an inserted computation of E. Some redundancy edges may not correspond to use-def edges for the temporary; such an edge either represents redundancy that cannot safely be eliminated or has the property that the expression value turns out to be available at both the head and the tail of the edge.*

Therefore, we can imagine that the task of our PRE algorithm is to begin by determining the set of redundancy edges for each expression, and then refining these edge sets to form the use-def relation for each expression's temporary variable. Notice that the use-def relation for an expression's temporary tells us everything about how to transform the program: use points are those points where we replace a computation of the expression with a use of the temporary, and definition points are places where we compute the expression's value and save it to the temporary.

By closely connecting the redundancy relation for an expression with the use-def relation for the temporary variable introduced by PRE for that expression, Observation 1 also implicitly connects the FRG for the expression to the SSA form for the optimized expression's temporary variable. This connection is the main property allowing our algorithm to efficiently produce its output in SSA form.

2.6 The FRG in SSAPRE

The analysis performed by our SSAPRE algorithm operates on the FRG for each of the expressions being optimized, so the algorithm incorporates a method for constructing the FRG. Because the FRG representation shares many of the characteristics of SSA form, the method to build the FRG closely parallels the standard SSA construction algorithm. The first two steps of the SSAPRE algorithm construct the FRG, with $\perp$ operands of Φ indicating those paths on which the expression is not evaluated. The first step, called *Φ-Insertion*, inserts Φ's at the iterated dominance frontier of each occurrence of E. In the second step, called *Rename*, we assign redundancy class numbers to occurrences of E according to the values they compute and their positions in the program.

The Φ's in the FRG serve as anchor points for placement analysis in PRE. Placement analysis involves two separate data flow analysis steps. The third step in SSAPRE, *DownSafety*, performs backward data flow propagation on the FRG to identify the Φ's that are down-safe. The fourth step, *WillBeAvail*, performs forward data flow propagation on the graph to predict the Φ's where the computation E will be made available following insertions for PRE. Using these data flow results, the fifth step, *Finalize*, can pinpoint the locations in the program where the computation is to be inserted. *Finalize* also identifies occurrences of E that are fully redundant taking into account the effects of these insertions, and refines the FRG to a form isomorphic to the SSA graph for t. At this point, the optimized output is completely determined, and is represented by the updated FRG. The last step, *CodeMotion*, transforms the code to form the optimized program. The temporary t is introduced to save and reuse the values of E corresponding to instances of redundancy eliminated by SSAPRE.

Although partial redundancy elimination is not among the optimizations treated by Choi *et al.* [Choi et al. 1991], much of our algorithm can be cast in their sparse data flow evaluation graph framework. Our FRG for each expression is a "flow graph" in their terminology, and the *DownSafety* and *WillBeAvail* steps of SSAPRE are examples of a class of procedures they call "evaluations."

3. SSAPRE ALGORITHM

In this section, we describe the SSAPRE algorithm. Like authors of earlier work [Rosen et al. 1988; Dhamdhere et al. 1992; Knoop et al. 1992; Drechsler and Stadel 1993], we assume all *critical edges* in the control flow graph have been removed by inserting empty basic blocks at such edges.[4] Breaking these edges allows us to model insertions as edge placements, even though we insert at the ends of the predecessor blocks. The idea of inserting basic blocks on critical edges to expand opportunities for safe code motion appears to have originated in [Rosen et al. 1988], or possibly [Drechsler and Stadel 1988]. [Dhamdhere 1988] proposed the related technique of splitting edges "on demand."

We assume the input is a program in SSA form. We assume prior computation of the dominator tree (DT) and iterated dominance frontiers (DF^+) with respect to the control flow graph of the program. These structures must already have been computed and used if the program was put into SSA form using the algorithm of [Cytron et al. 1991]. Finally, we make the following two simplifying assumptions about the input SSA program:

(1) Each ϕ assignment has the property that its left-hand side and all of its operands are versions of the same original program variable; and

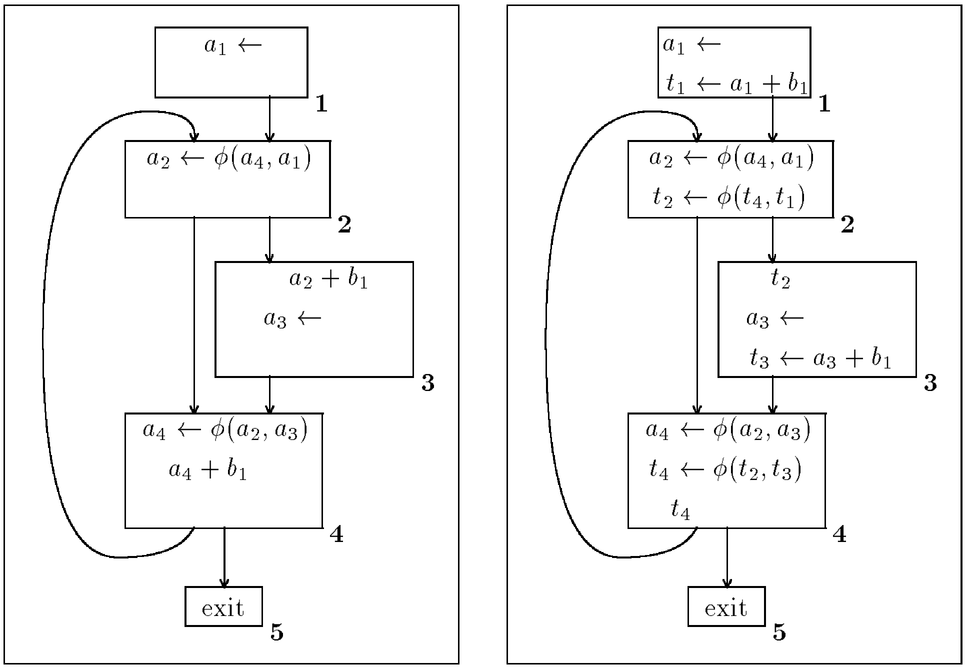
(2) The live ranges of different versions of the same original program variable do not overlap.

These assumptions are guaranteed to hold immediately after a program is put into SSA form [Cytron et al. 1991], and each of them can be relaxed at the cost of more difficult presentation and implementation of our algorithm. The interested reader is invited to investigate the changes involved in relaxing these assumptions.

We assume all expressions are represented as trees with leaves that are either constants or SSA-renamed variables. SSAPRE is applied to each lexically identified expression[5] independently, regardless of subtree nesting relationships. In Section 5, we describe a strategy that exploits the nesting relationship in expression trees to obtain greater optimization efficiency under SSAPRE. Indirect loads are also candidates for SSAPRE, but since they reference memory and can have aliases, the

[4] A critical edge is one whose tail block has multiple successors and whose head block has multiple predecessors.

[5] Computations belong to the same *lexically identified expression* if they apply exactly the same operator to exactly the same operands; the SSA versions of the variables are ignored in identifying expressions. For example, $a_1 + b_1$ and $a_2 + b_2$ are lexically identical forms, so they are instances of the same lexically identified expression.



(a) before optimization

(b) after optimization

Fig. 3. Example Program P (in SSA form) Before and After Optimization

indirect variables have to be in SSA form in order for SSAPRE to handle them. Using the HSSA form presented in [Chow et al. 1996] allows SSAPRE to uniformly handle indirect loads together with other expressions in the program.

In our description of the base algorithm, the initial SSA construction steps for expressions, Φ-*Insertion* and *Rename*, work on all expressions in the program simultaneously while passing through the entire program. The remaining steps of the algorithm can be efficiently applied to each expression separately. In Section 5, we describe an alternative scheme that allows all six steps of the algorithm to be applied to each lexically identified expression separately.

We use the program shown in Figure 3 as a running example to illustrate the various steps, and we call it Program P to distinguish it from additional examples that are interspersed to illustrate situations that do not appear in Program P.[6] Our examples assume we are working on the expression $a + b$ in the program.

Our presentation of SSAPRE is organized according to the six steps of the algorithm. As we describe each step, we also state and prove various lemmas which we use in establishing the theorems about SSAPRE in Section 4.

Before we present the details of SSAPRE, we establish two items of notation. We will use the notation $E^{(j)}$ to refer to the j^{th} lexically identified expression, and we will denote by $\mathcal{F}$ the set of Φ operators in the FRG for expression $E^{(j)}$. We omit

[6]For simplicity and compactness, we show the control flow graphs for our examples *with* critical edges; the examples are chosen so that breaking these edges would make no material difference.

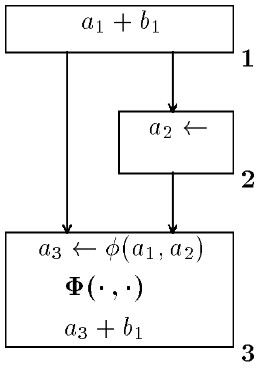


Fig. 4. Φ insertion due to ϕ for an expression operand

any index from the symbol $\mathcal{F}$; no confusion will result because the steps of SSAPRE referring to $\mathcal{F}$ handle a single expression at a time, so $\mathcal{F}$ will be understood to refer to the current expression under consideration.

3.1 The Φ-*Insertion* Step

A Φ for an expression is needed whenever different values of the same expression reach a common point in the program. There are two different situations that cause Φ's for expressions to be placed:

First, when an expression appears, we insert a Φ at its iterated dominance frontier (DF^+) as in [Cytron et al. 1991], because the occurrence may come to correspond to a definition of that expression's temporary.

The second situation that causes insertion of Φ's is when there is a ϕ for any variable contained in the expression, because such a ϕ indicates that an alteration of the expression reaches the merge point. In Figure 4, the Φ at block 3 is caused by the ϕ for a in the same block, which in turn reflects the assignment to a in block 2. Figure 5 shows our running example program after the Φ-Insertion step. Both Φ's in that figure are justified by real occurrences of $a + b$ and by the presence of ϕ's for the variable a.

Other algorithms for SSA ϕ placement with linear time complexity can also be used to place Φ's [Johnson et al. 1994; Sreedhar and Gao 1995]. We adapt the algorithm from [Cytron et al. 1991] because it is easier to understand and implement.

To make the details of the following lemma precise, we establish the following definition. Intuitively it is intended to capture the set of points in the program where the "current value" of an expression may change.

Definition 5. *An* evaluation *of expression $E^{(i)}$ is one of the following:*

—*a real occurrence of $E^{(i)}$;*

—*a Φ occurrence for $E^{(i)}$;*

—*an assignment to an operand of $E^{(i)}$.*

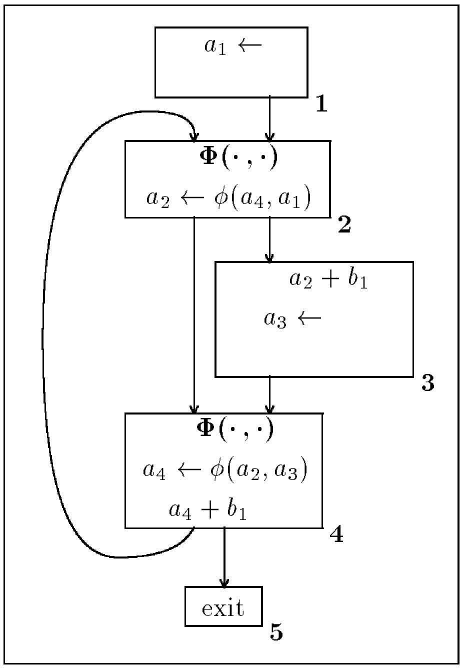


Fig. 5. Program P after Φ-Insertion

We say an evaluation of $E^{(i)}$ *reaches* a point p in the program if there is a path in the control flow graph from the evaluation to p that does not encounter any other evaluation of $E^{(i)}$. To distinguish assignments to expression operands from other evaluations, we say that assignments to operands of the expression have value $\perp$.

LEMMA 1. *(Sufficiency of Φ insertion) If B is a basic block where no Φ is inserted for the expression $E^{(i)}$, exactly one evaluation of $E^{(i)}$ can reach the entry to B.*

PROOF. Suppose two different evaluations of the expression, ψ_1 and ψ_2, reach the entry to B. It cannot be the case that ψ_1 and ψ_2 both dominate B; suppose without loss of generality that ψ_1 does not dominate B. Now there exists a block B_0 that dominates B, is reached by ψ_1 and ψ_2, and lies in $\mathrm{DF}^+(\psi_1)$ (*n.b.*, B_0 may be B). If ψ_1 is a real occurrence or a Φ, the Φ-*Insertion* step must have placed a Φ in B_0, contradicting the proposition that ψ_1 reaches B. If on the other hand ψ_1 is an assignment to an operand ν of the expression (so $\perp$ is among the values reaching B), there must be a ϕ for ν in B_0 by the correctness of the input SSA form. Hence Φ-*Insertion* must have placed a Φ in B_0, once again contradicting the proposition that ψ_1 reaches B. □

Section 5.3 describes a more efficient implementation that omits some Φ operators promised by Lemma 1, but that are unnecessary and could not participate in any optimization because the corresponding expressions are not partially anticipated where the omissions occur.

3.2 The *Rename* Step

The *Rename* step assigns redundancy class numbers to expression occurrences. The redundancy class numbering has the following two important properties. First, occurrences that have identical class numbers have identical values. Second, any control flow path that includes two different class numbers for some expression must cross an assignment to an operand of the expression or a Φ.

We apply the SSA Renaming algorithm as given in [Cytron et al. 1991], in which we conduct a preorder traversal of the dominator tree, but with the following modification. In addition to a renaming stack for each variable in the program, we maintain a renaming stack for every expression; entries on these expression stacks are popped as our dominator tree traversal backtracks past the blocks that contain them. Maintaining the variable and expression stacks together allows us to decide efficiently whether two occurrences of an expression should be given the same redundancy class number.

There are three kinds of occurrences of expressions in the program: (1) the occurrences in the original program, which we call *real* occurrences; (2) the Φ's inserted in the Φ-*Insertion* step; and (3) Φ operands, which are regarded as occurring at the ends of the predecessor blocks along the corresponding edges. The *Rename* algorithm performs the following steps upon encountering an occurrence q of the expression $E^{(i)}$. If q is a Φ, we assign q a new class number. Otherwise, we check the current version of every variable in $E^{(i)}$ (*i.e.*, the version on the top of each variable's rename stack) against the version of the corresponding variable in the occurrence on the top of $E^{(i)}$'s rename stack. If all the variable versions match, we assign q the same class as the top of $E^{(i)}$'s stack and record the upward edge between q and its representative occurrence by writing a reference to the representative occurrence in the field $def(q)$. If any of the variable versions does not match, we have two cases: (a) if q is a real occurrence, we assign q a new class number; (b) if q is a Φ operand, we assign the special class $\perp$ to that Φ operand to denote that the value of $E^{(i)}$ is unavailable at that point. Finally, we push q on $E^{(i)}$'s stack and proceed. Figure 6 shows the initial graph formed after our example has been renamed. The nodes in the FRG are annotated with their assigned redundancy class numbers in square brackets.

LEMMA 2. *(Correctness of renaming) If two occurrences of the same expression are assigned to the same class by* Rename*, the expression has the same value at those two occurrences.*

PROOF. This lemma follows directly from the fact that the *Rename* step assigns two occurrences of an expression to the same class only if all the SSA versions of their expression operands match. We appeal to the single-assignment property and the correctness of the SSA renaming algorithm for variables [Cytron et al. 1991] to complete the proof. □

LEMMA 3. *(Assigned classes capture all the redundancy) If two occurrences E_x, E_y are assigned class numbers x, y by* Rename*, exactly one of the following holds:*

—*no control flow path can reach from E_x to E_y without passing through a real (*i.e.*, non-ϕ) assignment to an operand of the expression (meaning that there is no redundancy between the occurrences); or*

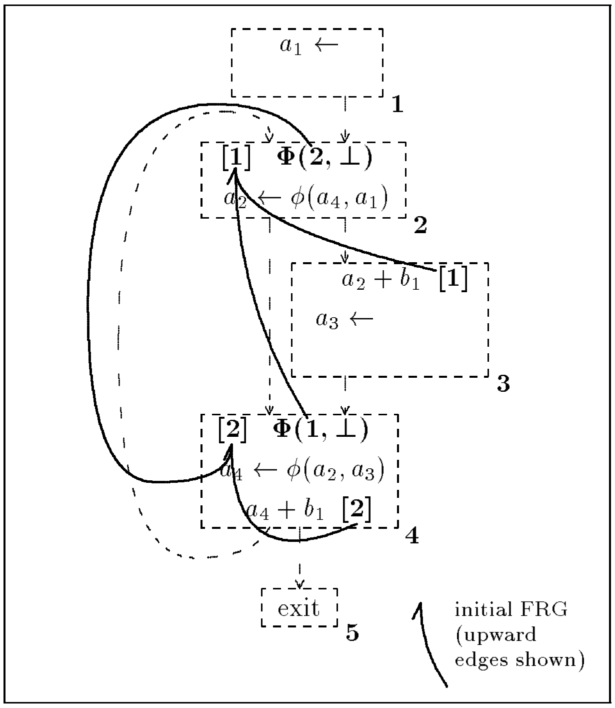


Fig. 6. The initial FRG for $a + b$ in Program P

—*there is a path (possibly empty, in which case $x = y$) of upward edges in the FRG from the representative of class y to the representative of class x (implying that the redundancy between E_x and E_y is exposed to the algorithm).*

PROOF. Suppose there is a control flow path $\mathcal{P}$ from E_x to E_y that does not pass through any assignment to an operand of the expression. Our proof will proceed by induction on the number of Φ's for the expression traversed by $\mathcal{P}$.

If $\mathcal{P}$ encounters no Φ, $x = y$ establishing the basis for our induction. If $\mathcal{P}$ hits at least one Φ, the last Φ on $\mathcal{P}$ defines E_y. Now we apply the induction hypothesis to that part of $\mathcal{P}$ up to the corresponding operand of that Φ. □

To save space, we do not prove that the object constructed by the Φ-*Insertion* and *Rename* steps fulfills the definition of FRG given in Section 2.3. We leave that proof as a straightforward exercise for the interested reader because it is not required to establish the correctness of our algorithm.

Section 5.4 describes in detail an implementation of the *Rename* step that derives greater efficiency by more thoroughly exploiting the SSA form of the input program.

3.3 The *DownSafety* Step

One criterion required for PRE to insert a computation is that the computation is down-safe (or anticipated) at the point of insertion [Kennedy 1972; Morel and Renvoise 1979; Knoop et al. 1994]. This condition serves to ensure both that inserted computations do not introduce exceptions to paths that lacked them before

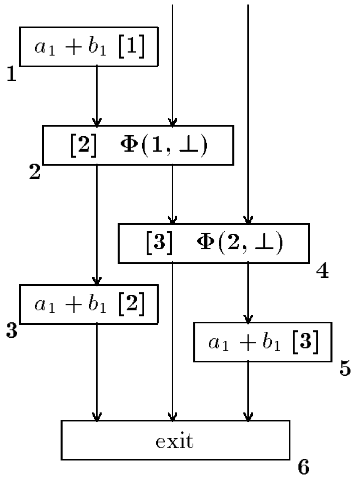


Fig. 7. Propagation in the *DownSafety* step

optimization, and that inserted computations do not introduce new redundancy to the program. In the FRG constructed by *Rename*, each node either represents a real occurrence of the expression or is a Φ. It can be shown that SSAPRE insertions are necessary only at Φ's, so down-safety needs to be computed only at points where Φ's appear. Using the factored redundancy graph, down-safety can be sparsely computed by propagation along the upward edges.

A Φ is not down-safe if there is a control flow path from that Φ along which the expression is not evaluated before program exit or before being altered by redefinition of one of its variables. Except for loops with no exit, this can happen only due to one of the following cases: (a) there is a path to exit along which the Φ's redundancy class does not occur; or (b) there is a path to exit along which the only occurrence of the Φ's redundancy class is as an operand of a Φ that is not down-safe. Case (a) represents the initialization for our backward propagation to compute down-safety; all other Φ's are initially marked *down_safe*. *DownSafety* propagation is based on case (b): beginning at each Φ that is initially not marked *down_safe*, the algorithm searches along upward edges that do not traverse any real occurrence of the expression, clearing the *down_safe* flag for each Φ visited. Since traversing a real occurrence of the expression blocks the propagation, the algorithm assumes each Φ operand is marked with a flag *has_real_use* that is true when the path to the Φ operand from its representative occurrence crosses a real occurrence of the same redunancy class.

It is convenient to perform initialization of the case (a) *down_safe* and computation of the *has_real_use* flags during a dominator-tree preorder pass over the FRG. Since *Rename* conducts such a pass, we can include these calculations in the *Rename* step with minimal overhead. Initially, all *down_safe* flags are true and all *has_real_use* flags are false. When *Rename* assigns a new class to a real occurrence of expression $E^{(i)}$, sets an operand of a Φ for $E^{(i)}$ to $\perp$, or encounters a program

```
procedure Reset_downsafe(X)
  if (has_real_use(X) or def(X) is not a Φ)
    return
  f ← def(X)
  if (not down_safe(f))
    return
  down_safe(f) ← false
  for each operand ω of f do
    Reset_downsafe(ω)
end Reset_downsafe

procedure DownSafety
  for each f ∈ 𝓕 do
    if (not down_safe(f))
      for each operand ω of f do
        Reset_downsafe(ω)
end DownSafety
```

Fig. 8. Algorithm for *DownSafety*

exit, it examines the occurrence on the top of $E^{(i)}$'s stack before pushing the current occurrence. If the top of stack is a Φ occurrence, *Rename* clears that Φ's *down_safe* flag because the class it represents does not occur along the path to the current occurrence (or exit). When *Rename* assigns a class to a Φ operand, it sets that operand's *has_real_use* flag if and only if a real occurrence in the same class appears at the top of the rename stack.

In the example of Figure 7, the Φ in block 4 is marked not *down_safe* during initialization by the *Rename* step. The *DownSafety* step propagates a false value for *down_safe* to the Φ in block 2 along the upward edge between the appearance of class 2 as an operand of the Φ in block 4 and its definition by the Φ in block 2. Figure 8 gives the *DownSafety* propagation algorithm. In our running example program (Figure 6), both Φ's are down-safe.

LEMMA 4. *(Correctness of down_safe) A Φ is marked down_safe after DownSafety if and only if the expression is fully anticipated at that Φ.*

PROOF. We first note that each Φ marked not *down_safe* during *Rename* is indeed not down-safe. The SSA renaming algorithm has the property that every definition dominates all its uses. Suppose that a Φ appears on the top of the stack when *Rename* creates a new class for a real occurrence or a Φ operand or encounters a program exit. In the case where a program exit is encountered, the Φ is obviously not down-safe because there is a path in the dominator tree from the Φ to exit containing no use of the Φ. Similarly, if *Rename* assigns a new class to a real occurrence, it does so because some expression operand ν has a different version in the current occurrence from its version at the Φ. Therefore there exists a path in the dominator tree from the Φ to the current occurrence along which there is an assignment to ν. By Lemma 1, at least one such assignment is a real assignment (not a ϕ). Hence the expression is not fully anticipated at the Φ on the top of the stack.

Next we make the observation that any Φ f whose *down_safe* flag gets cleared during the *DownSafety* step is not down-safe, since there is a path of upward edges in the FRG from a Φ that is not down-safe to f, where no edge in the path crosses any real use of the expression value. Indeed one such path appears on the recursion stack of the *Reset_downsafe* procedure at the time the *down_safe* flag is cleared.

Finally, we need to show that all the Φ's that are not down-safe are so marked at the end of *DownSafety*. This fact is a straightforward property of the depth-first search propagation performed by *Reset_downsafe*. □

3.4 The *WillBeAvail* Step

The *WillBeAvail* step has the task of predicting whether the expression will be available at each Φ occurrence following insertions for PRE. In the *Finalize* step, insertions will be performed at incoming edges corresponding to Φ operands at which the expression will not be available (without that insertion), but the Φ's *will_be_avail* predicate is true. *WillBeAvail* begins by computing the set of Φ occurrences where the expression value can safely be made available. Next, *WillBeAvail* effectively computes the set of Φ's where the expression value must be available in any computationally optimal placement; it is exactly these Φ's where the expression will be made available by SSAPRE, and the resulting placement minimizes the live ranges of the introduced expression temporary.

The *WillBeAvail* step consists of two forward propagation passes performed sequentially, in which we conduct simple reachability search in the FRG for each expression. The first pass computes the *can_be_avail* predicate for each Φ by first initializing it to true for all Φ's. It then begins with the "boundary" set of Φ's at which the expression cannot be made available by any down-safe set of insertions. These are Φ's that do not satisfy the *down_safe* predicate and have at least one $\perp$-valued operand. The *can_be_avail* predicate is set to false for every such Φ and the false value is propagated from these nodes to others that are not down-safe and that are reachable along downward FRG edges, excluding edges at which *has_real_use* is true. After this propagation step, *can_be_avail* is false for a Φ if and only if no down-safe placement of computations can make the expression available.

The Φ's where *can_be_avail* is true together designate the range of down-safe program areas for insertion of the expression, plus areas that are not down-safe but where the expression is fully available in the original program.[7]

The second pass works within the region computed by the first pass to determine the Φ's where the expression will be available following the insertions we will actually make, which implicitly determines the *latest* (and final) insertion points. This pass computes the information responsible for minimizing the live ranges of the introduced expression temporary, and is analogous to the computation of the predicate LATERIN in [Drechsler and Stadel 1993]. It works by propagating the *later* predicate, which it initializes to true wherever *can_be_avail* is true. It then begins with the Φ operands corresponding to real occurrences of the expression in the program, and propagates a false value for *later* forward to those points beyond which insertions cannot be postponed (moved downward) without introducing

[7]The entry points to this region (the $\perp$-valued Φ operands) can be thought of as SSAPRE's *earliest* insertion points. These may be later than the earliest insertion points in [Knoop et al. 1992] and [Drechsler and Stadel 1993] because their bit-vector schemes allow earliest insertion at non-merge blocks.

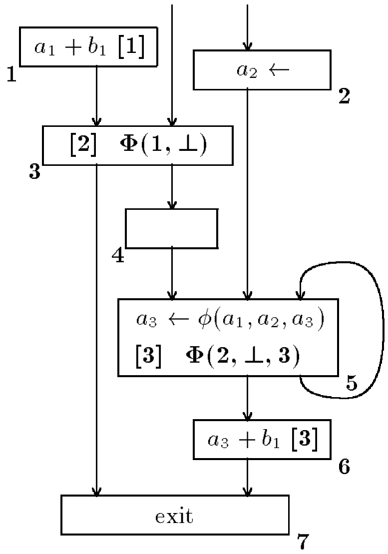


Fig. 9. Example showing the role of *later*

unnecessary new redundancy.[8]

At the end of the second pass, *will_be_avail* for a Φ is given by:

$$\textit{will\_be\_avail} = \textit{can\_be\_avail} \land \neg \textit{later}.$$

In the example program of Figure 9, the Φ in block 5 satisfies *down_safe*, *can_be_avail*, and *later*. Therefore, although the expression value could safely be made available by insertions in blocks 2 and 4, the *later* predicate prevents such insertion, which would eliminate no redundancy and would unnecessarily extend the live range of the expression temporary. In our running example (Figure 6), both Φ's satisfy *will_be_avail*.

For convenience, we define a predicate to indicate those Φ operands where we will perform insertions: We say *insert* holds for a Φ operand if and only if the following hold:

—the Φ satisfies *will_be_avail*; and

—the operand is $\perp$, or *has_real_use* is false for the operand and the operand is defined by a Φ that does not satisfy *will_be_avail*.

Figure 10 gives the *WillBeAvail* propagation algorithms.

Recall that the term *placement* refers to the set of points in the program where a particular expression's value is computed.

LEMMA 5. *(Correctness of can_be_avail) A Φ satisfies can_be_avail if and only if some safe placement of computations makes the expression available immediately*

[8]The result is that those Φ's satisfying *later* are exactly those that are *can_be_avail* but not reachable from any real occurrence along downward FRG edges.

```
procedure Reset_can_be_avail(g)
  can_be_avail(g) ← false
  for each f ∈ F with operand ω with g = def(ω) do
    if (not has_real_use(ω)) {
      if (not down_safe(f) and can_be_avail(f))
        Reset_can_be_avail(f)
    }
end Reset_can_be_avail

procedure Compute_can_be_avail
  for each f ∈ F in the program do
    can_be_avail(f) ← true
  for each f ∈ F in the program do
    if (not down_safe(f) and
        can_be_avail(f) and
        ∃ an operand of f that is ⊥)
      Reset_can_be_avail(f)
end Compute_can_be_avail

procedure Reset_later(g)
  later(g) ← false
  for each f ∈ F with operand ω with g = def(ω) do
    if (later(f))
      Reset_later(f)
end Reset_later

procedure Compute_later
  for each f ∈ F do
    later(f) ← can_be_avail(f)
  for each f ∈ F do
    if (later(f) and
        ∃ an operand ω of f such that
          (def(ω) ≠ ⊥ and has_real_use(ω)))
      Reset_later(f)
end Compute_later

procedure WillBeAvail
  Compute_can_be_avail
  Compute_later
end WillBeAvail
```

Fig. 10. Algorithm for *WillBeAvail*

after the Φ*.*

PROOF. Let $f \in \mathcal{F}$ be a Φ satisfying *can_be_avail*. If f satisfies *down_safe*, the result is immediate because it is safe to insert computations of the expression at each of f's operands. If f is not down-safe and satisfies *can_be_avail*, note that the expression is available in the unoptimized program at f because there is no path to f from a Φ with a $\perp$-valued operand along downward edges lacking *has_real_use* in the FRG.

Now let $f \in \mathcal{F}$ be a Φ that does not satisfy *can_be_avail*. When the algorithm reset this *can_be_avail* flag, the recursion stack of *Reset_can_be_avail* gave a path bearing witness to the fact that no safe set of insertions can make the expression available at f. $\square$

Lemma 6. *(Correctness of later) A can_be_avail Φ satisfies later after WillBeAvail if and only if there exists a computationally optimal placement under which the expression value is not available immediately after the Φ.*

Proof. By inspection of the *Compute_later* algorithm, the set of *can_be_avail* Φ's not satisfying *later* after *WillBeAvail* is exactly the set of *can_be_avail* Φ's reachable along downward edges in the FRG from a *can_be_avail* Φ with an operand satisfying *has_real_use*. Let $\mathcal{P}$ be a path of downward edges in the FRG from a *can_be_avail* Φ with an operand satisfying *has_real_use* to a given $f \in \mathcal{F}$ with $later(f)$ = false and $can\_be\_avail(f)$ = true. We will prove by induction on the length of $\mathcal{P}$ that f must be made available by any computationally optimal placement.

If f is not down-safe, the fact that f is *can_be_avail* means all of f's operands must be fully available in the unoptimized program. They are therefore trivially available under any computationally optimal placement, making the result of f available as well.

In the case where f is down-safe, if $\mathcal{P}$ contains no edges there is a *has_real_use* operand of f. Such an operand must be fully available in the optimized program, so any insertion below f would be redundant with respect to the real occurrence(s) corresponding to that operand, contradicting computational optimality. Since f is down-safe, there already exist real occurrences in the unoptimized program that are redundant with respect to the real occurrences corresponding to the operand, and any computationally optimal placement must eliminate that redundancy. The way to accomplish this is to perform insertions that make the expression fully available at f.

If f is down-safe and $\mathcal{P}$ contains at least one edge, we apply the induction hypothesis to the Φ defining the operand of f corresponding to the final edge on $\mathcal{P}$ to conclude that that operand must be made available by any computationally optimal placement. As a consequence, any computationally optimal placement must make f available by the same argument as in the basis step (previous paragraph). $\square$

The following lemma shows that the *will_be_avail* predicate computed by *WillBeAvail* faithfully corresponds to availability in the program after insertions are performed for Φ operands satisfying *insert*.

Lemma 7. *(Correctness of will_be_avail) The union of the set of insertions chosen by SSAPRE with the set of real occurrences makes the expression available immediately after a Φ if and only if that Φ satisfies will_be_avail.*

Proof. We establish the "if" direction with a simple induction proof showing that if there is some path leading to a particular Φ in the optimized program along which the expression is unavailable, that Φ does not satisfy *will_be_avail*. Let $Q(k)$ be the following proposition:

> For any $f \in \mathcal{F}$, if there is a path $\mathcal{P}(f)$ of k downward edges in the FRG beginning with $\perp$, passing only through Φ's along edges that do not satisfy *has_real_use* $\vee$ *insert*, and ending at f, f is not *will_be_avail*.

$Q(0)$ follows directly from the fact that f has a $\perp$-valued operand and no insertion is performed for any operand of f, so f is not marked *will_be_avail*. The fact that f

has a $\perp$-valued operand implies that such an insertion would be required to make f available.

Now to see $Q(k)$ for $k > 0$, notice that $Q(k-1)$ implies that the operand of f corresponding to the final edge of $\mathcal{P}(f)$ is defined by a Φ that is not *will_be_avail*, and there is no real occurrence of the expression on the control flow path from that defining Φ to the operand of f. Since we do not perform an insertion for that operand, f cannot satisfy *will_be_avail*.

To establish the "only if" direction, suppose $f \in \mathcal{F}$ does not satisfy *will_be_avail*. Either f does not satisfy *can_be_avail* or f satisfies *later*. In the former case, f is not available in the optimized program because the insertions performed by SSAPRE are down-safe. In the latter case, f was not processed by *Reset_Later*, meaning that it is not reachable along downward edges from a Φ satisfying *will_be_avail*. Therefore, insertion above f would be required to make f's result available, but f is not *will_be_avail* so the algorithm performs no such insertion. □

3.5 The *Finalize* Step

The *Finalize* step plays the role of transforming the factored redundancy graph to the optimized form that reflects insertions and in which no Φ operand is $\perp$. The *Finalize* step consists of two parts, *Finalize_1* and *Finalize_2*. *Finalize_1* performs the following tasks:

—Each real occurrence of the expression is marked with a flag called *reload* to indicate whether it should be computed on the spot or reloaded from the temporary.

—For Φ's where *will_be_avail* is true, insertions are performed at the incoming edges that correspond to Φ operands at which the expression is not available.

—Φ's whose *will_be_avail* predicate is true may become ϕ's for t. Φ's that are not *will_be_avail* will not be part of the SSA form for t, and FRG edges from *will_be_avail* Φ's that reference them are updated to refer to other (real or inserted) occurrences.

—The FRG structure is updated to reflect the factored use-def relation for the expression temporary in the optimized program. This restructuring is accomplished by resetting the *def* field of each operand of a Φ satisfying *will_be_avail* and each real occurrence that will be reloaded from the temporary so that these *def* fields refer to the expression occurrences that will become the definitions of the corresponding SSA versions of the temporary.

The following tasks are the responsibility of *Finalize_2*:

—Each real occurrence that is not reloaded from the temporary is marked with a *save* flag according as the expression value should be saved to the temporary.

—Extraneous Φ's are removed.

Finalize_1 creates a table *Avail_def* (for *available definitions*) to perform the first three of the above tasks. The indices into this table are the redundancy class numbers for the current expression. *Avail_def*$[x]$ will point to the expression occurrence that defines the value of occurrences in redundancy class x when each reload of class x is seen. This defining occurrence must be either: (a) a real occurrence, or (b) a Φ for which *will_be_avail* is true. *Finalize_1* performs a preorder traversal of the

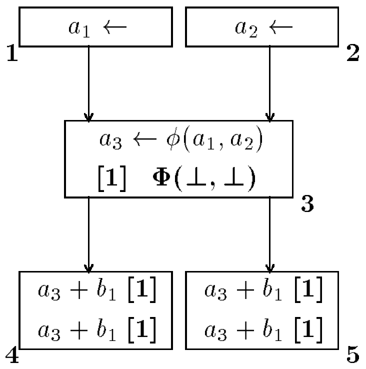


Fig. 11. Example showing two available definitions for redundancy class 1

dominator tree of the program control flow graph. In the course of this traversal it will visit each representative occurrence whose value will be saved to a version of the temporary, t_y, before it visits the occurrences that will reference t_y; such a reference is either: (a) a redundant computation that will be replaced by a reload of t_y, or (b) a use of class x as a Φ operand that will become a use of t_y as a ϕ operand. Although the processing order of *Finalize* is modeled after the standard SSA rename step [Cytron et al. 1991], *Finalize* does not require any renaming stack because SSA versions have already been assigned and only limited changes can be needed.

Initially all the entries of *Avail_def* are $\perp$. In the course of its traversal, *Finalize* will process occurrences as follows:

(1) Φ — If *will_be_avail* is false, nothing needs to be done, since this Φ will not figure in the SSA form for the real temporary. Otherwise, we must be visiting class x for the first time; we set *Avail_def*$[x]$ to this Φ.

(2) Real occurrence — If *Avail_def*$[x]$ is $\perp$, we are encountering for the first time a point where a value of occurrences in class x will be available. If *Avail_def*$[x]$ is set to an occurrence that does not dominate the current occurrence, the current occurrence is also a definition of class x. Figure 11 shows how this situation can arise: we have a Φ in block 3 that does not satisfy *will_be_avail*, so each branch along which the Φ's class number is used must have its own available definition and its own store to the expression temporary. The class represented by the Φ will therefore correspond to two different versions of the expression temporary t in our example. If *Avail_def*$[x]$ is either $\perp$ or an occurrence that does not dominate the current occurrence, we update *Avail_def*$[x]$ to the current occurrence. Otherwise, the current occurrence X is a use of an available value for class x, and we set the *reload* flag for X and record the value of *Avail_def*$[x]$ in *def*(X).

(3) Operand of Φ in a successor block[9] — If *will_be_avail* of the Φ is false, nothing

[9]Recall that Φ operands are considered as occurring at their corresponding predecessor blocks.

```
procedure Finalize_1
  for each redundancy class x of the current expression do
    Avail_def[x] ← ⊥
  for each occurrence X of the current expression in preorder DT traversal order do {
    x ← class(X)
    if (X is a Φ occurrence) {
      if (will_be_avail(f))
        Avail_def[x] ← f
    }
    else if (X is a real occurrence)
      if (Avail_def[x] is ⊥ or
          Avail_def[x] does not dominate X) {
        reload(X) ← false
        Avail_def[x] ← X
      }
      else {
        reload(X) ← true
        def(X) ← Avail_def[x]
      }
    else { /* X is a Φ operand occurrence */
      let f be the Φ in the successor block of this operand
      if (will_be_avail(f))
        if (X satisfies insert) {
          insert the current expression at the end of the block containing X
          def(X) ← inserted occurrence
        }
        else
          def(X) ← Avail_def[x]
    }
  }
end Finalize_1
```

Fig. 12. Algorithm for the first part of *Finalize*

needs to be done. Otherwise if the operand X satisfies *insert*, we insert a computation of the current expression at the end of the current block and set *def*(X) to refer to the inserted computation. If X does not satisfy *insert*, we set *def*(X) to refer to the current available definition for the redundancy class of X.

The full algorithm for *Finalize_1* is given in Figure 12.

To determine those real occurrences that must be saved to the temporary, *Finalize_2* performs a backward search over the FRG. The search begins at the set of real occurrences that are marked *reload* and progresses backward along upward edges using the *def* field for each reloaded real occurrence as set by *Finalize_visit* during *Finalize_1*. Every real occurrence that defines a Φ operand or real occurrence encountered in the search must be computed and saved to the temporary, so the *save* flag for each such occurrence is set.

The removal of extraneous Φ's, or FRG minimization, is not a necessary task as far as PRE is concerned. However, the extraneous Φ's take up space in the program representation, and may affect the efficiency of other SSA-based optimizations to be applied after PRE. Removing an extraneous Φ requires changing the occurrences

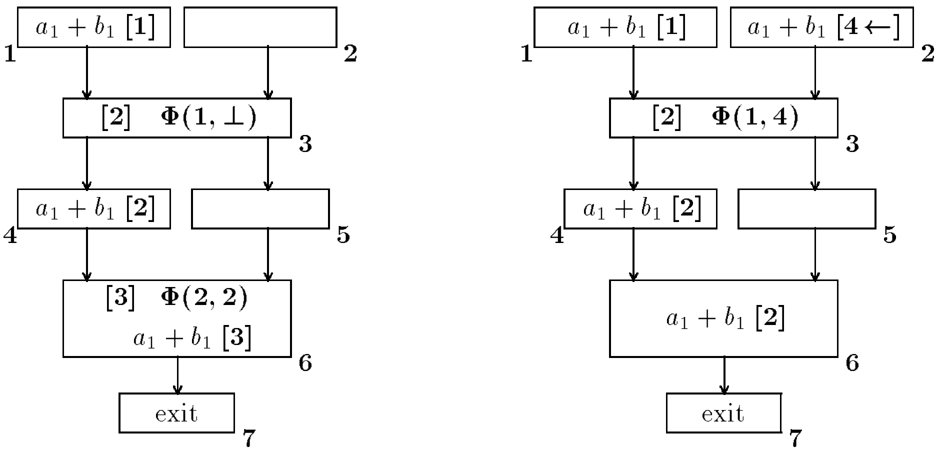
```
procedure Set_save(X)
  if (X is a real occurrence)
    save(X) ← true
  else if (X is a Φ occurrence)
    for each operand ω of X do
      if (not processed(ω))
        Set_save(def(ω))
  if (X is real or inserted)
    for each f ∈ F that is a will_be_avail Φ appearing in DF+(X) do
      extraneous(f) ← false
end Set_save

procedure Set_replacement(g, replacing_def)
  for each will_be_avail f ∈ F with j^th operand defined by g do
    if (extraneous(f))
      Set_replacement(f, replacing_def)
    else
      replace j^th operand of f by replacing_def
  for each real occurrence X satisfying reload with def(X) = g do
    def(X) ← replacing_def
  F ← F − {g}
end Set_replacement

procedure Finalize_2
  for each f ∈ F satisfying will_be_avail do
    extraneous(f) ← true
  for each real occurrence X do
    save(X) ← false
  for each f ∈ F do
    for each operand ω of f do
      processed(ω) ← false
  for each real occurrence X satisfying reload do
    Set_save(def(X))
  for each f ∈ F do
    if f satisfies will_be_avail {
      if (extraneous(f))
        for each operand ω of f do
          if ((def(ω) is a Φ and not extraneous(def(ω))) or
              (def(ω) is real) or
              (def(ω) is inserted))
            Set_replacement(f, def(ω))
    }
    else
      F ← F − {f}
end Finalize_2
```

Fig. 13. Algorithm for the second part of *Finalize*

in its redundancy class to refer to a different class that defines the value of the Φ. FRG minimization can be implemented as a variant of the ϕ insertion step in SSA construction [Cytron et al. 1991; Johnson et al. 1994; Sreedhar and Gao 1995]. We initially mark all the Φ's as being extraneous. Applying the ϕ insertion algorithm, we can find and mark the Φ's that are not extraneous based on the iterated dominance frontier of the set of real occurrences with the *save* flag set plus the inserted computations. We then pass over all the extraneous Φ's to determine



(a) before *Finalize* (b) after *Finalize*

Fig. 14. Effect of FRG minimization

a *replacing class* for each one. Whenever an extraneous Φ represents redundancy class x and has an operand using redundancy class y that is not defined by an extraneous Φ, y is the replacing class for x. From such a Φ we propagate the replacing class along downward edges: once the replacing class for a Φ is known, the replacing class for every occurrence defined by that Φ becomes known (the replacing class for each such occurrence is the same as the replacing class of the Φ). See Figure 13. It is straightforward to see that this method correctly replaces all references to extraneous Φ's by references to non-extraneous occurrences.

The effect of FRG minimization in the *Finalize* step can be seen in Figure 14, where the algorithm has discovered that the value of $a + b$ in block 4 should be reloaded from the expression temporary, so the Φ in block 6 is extraneous. Figure 14(b) shows the form of the program after FRG minimization removes the extraneous Φ and updates the real occurrence in block 6 that referred to it.

Figure 15 shows our example program P at the end of the *Finalize* step. The new classes 3 and 4 have been introduced, defined by inserted computations in blocks 1 and 3, respectively, and each of the real occurrences from the original program will be reloaded from the temporary.

LEMMA 8. *(Correctness of save/reload) At the point of any reload, the temporary contains the value of the expression.*

PROOF. This lemma follows directly from the *Finalize* algorithm and from the fact that *Rename* assigns redundancy classes to occurrences while traversing the FRG in dominator-tree preorder. In particular, *Finalize* ensures directly that each reload is dominated by its available definition. Because the live ranges of different redundancy classes do not overlap, each reloaded occurrence must refer to its available definition. □

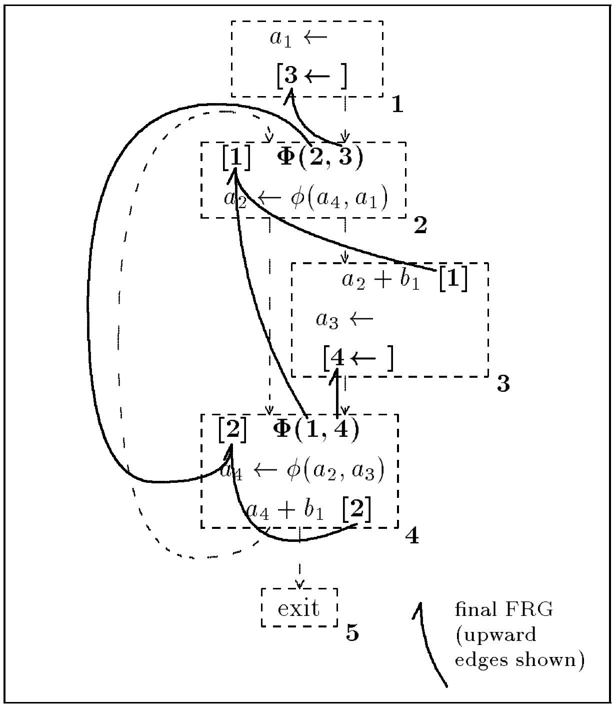


Fig. 15. Program P after *Finalize*

LEMMA 9. *(Optimality of reload) The optimized program does not compute the expression at any point where it is fully available.*

PROOF. It is straightforward to check that the optimized program reloads the expression value for any occurrence defined by a Φ satisfying *will_be_avail*, and it reloads the expression value for any occurrence dominated by another real occurrence in the same class. Therefore we need only note that *will_be_avail* accurately reflects availability in the optimized program (by Lemma 7) and that by the definition of *insert* we only insert for Φ operands where the insertion is required to achieve availability. □

3.6 The *CodeMotion* Step

Once the factored redundancy graph has been processed by *Finalize*, the only remaining task is to update the SSA program representation to reflect the results of PRE. This involves introducing the expression temporary t for the purpose of eliminating redundant computations.

The *CodeMotion* step walks over the FRG in dominator-tree preorder. At a real occurrence, if *save* is true, it generates a save of the result of the computation into a new version of t. For Φ operand occurrences and real occurrences with the *reload* flag set, it replaces the computation by a use of t whose SSA version is determined by the version already assigned to the present occurrence's representative occurrence. At an inserted occurrence, it saves the value of the inserted computation into a new

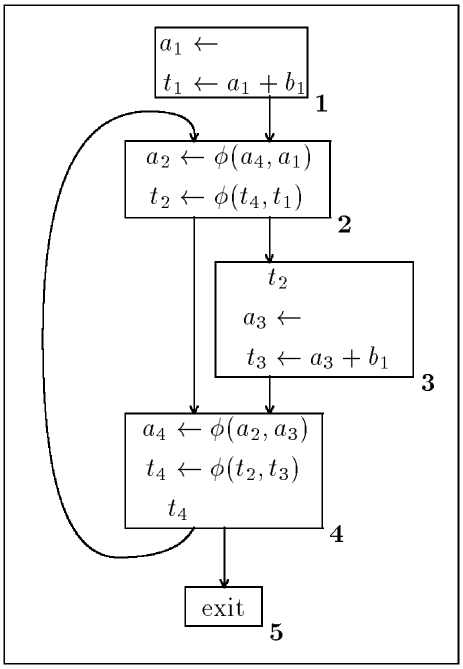


Fig. 16. Program P after *CodeMotion*

version of t. At each Φ, it generates a corresponding ϕ for t. Figure 16 shows our running example program at the end of the *CodeMotion* step.

4. THEORETICAL RESULTS

In this section we derive our main results about SSAPRE from the lemmas already given.

THEOREM 1. *SSAPRE chooses a* safe *placement of computations;* i.e., *along any path from entry to exit exactly the same values are computed in the optimized program as in the original program.*

PROOF. Since insertions take place only at points satisfying *down_safe*, this theorem follows directly from Lemma 4. □

THEOREM 2. *SSAPRE generates a reload of the correct expression value from the temporary at a real occurrence point if and only if the expression value is available at that point in the optimized program.*

PROOF. This theorem follows from the fact that a reload is generated for a real occurence if and only if it is dominated by a *will_be_avail* Φ of the same class (in which case we appeal to Lemma 7 for the availability of the expression at the reload point), or by a real occurrence of the same class that is marked *save* by *Finalize*. □

THEOREM 3. *SSAPRE generates a save to temporary at a real occurrence or insertion point if and only if the following hold:*

—*the expression value is unavailable (in the optimized program) just before that point, and*
—*the expression value is partially anticipated just after that point (*i.e.*, there will be a use of the saved value).*

Proof. This theorem follows directly from Lemma 9 and from the fact that the *Finalize* algorithm sets the *save* flag for a real occurrence only when there is a control flow path from that occurrence to an occurrence where the *reload* flag is set, with no intervening *save*. □

Theorem 4. *SSAPRE chooses a* computationally optimal *placement;* i.e.*, no safe placement can result in fewer evaluations of the expression along any path from entry to exit in the control flow graph.*

Proof. We need only show that any redundancy remaining in the optimized program cannot be eliminated by any safe placement of computations. Suppose $\mathcal{P}$ is a control flow path in the optimized program leading from one computation, ψ_1, of the expression to another computation, ψ_2, of the same expression with no assignment to any operand of the expression along $\mathcal{P}$. By Theorem 2, the expression value cannot be available just before ψ_2, so ψ_2 is not dominated by a real occurrence of the same class (by Lemma 9) nor is it defined by a *will_be_avail* Φ (by Lemma 7). Because there is no assignment to any expression operand along $\mathcal{P}$, the definition of ψ_2's class must lie on $\mathcal{P}$, and since it cannot be a real occurrence nor a *will_be_avail* Φ, it must be a Φ that is not *will_be_avail*. Such a Φ cannot satisfy *later* because one of its operands is reached by ψ_1, so it must not be down-safe. So no safe set of insertions could make ψ_2 available while eliminating a computation from $\mathcal{P}$. □

Theorem 5. *SSAPRE chooses a* lifetime-optimal *placement; specifically, if p is the point just after an insertion made by SSAPRE and $\mathcal{C}$ denotes any computationally optimal placement, $\mathcal{C}$ makes the expression fully available at p.*

Proof. This theorem is a direct consequence of Lemma 6 and Theorem 4. □

Theorem 6. *SSAPRE produces minimal SSA form for the generated temporary.*

Proof. This minimality result follows directly from the correctness and minimality of the dominance frontier ϕ-insertion algorithm [Cytron et al. 1991]. Each Φ remaining after *Finalize* is justified by being on the iterated dominance frontier of some real or inserted occurrence that will be saved to the temporary. □

5. PRACTICAL IMPLEMENTATION

In this section, we discuss some issues related to the efficient and practical implementation of SSAPRE in an optimizing compiler. An implementation can take advantage of the sparse approach in dramatically reducing the maximum storage needed to optimize all the expressions in the program. This can be accomplished by maintaining a worklist that contains the different lexically identified expressions that await processing by SSAPRE. In the absence of redundancy, we can exploit the nesting relationship in expression trees to avoid unnecessary work in the ancestral part of the tree. We also present more efficient forms of the algorithms for Φ-*Insertion*, *Rename*, and the computation of *save* in the *Finalize* step than the versions we presented in Section 3.

5.1 Worklist-driven PRE

The algorithms we presented in Section 3 for Φ-*Insertion* and *Rename* work on all expressions in the program simultaneously while passing through the entire program. Handling all expressions at once creates overhead in memory usage because the Φ's for all the expressions in the program need to be represented together, and the renaming stacks for all the expressions have to coexist in the *Rename* step. Some details of the issue of representing the FRG for an expression were not made explicit in Section 3. We now present a worklist-driven PRE approach that addresses these issues.

In the worklist-driven approach, we manage the lexically identified expressions that need to be worked on by PRE using a worklist. We add an initial pass, *Collect-Occurrences*, that scans the program to create the initial worklist. For each lexically identified expression, we represent its occurrences in the program by a set of *occurrence nodes*. Each occurrence node provides enough information to pinpoint the location of that occurrence in the program. *Collect-Occurrences* is the only pass that needs to look at the entire program. The six steps of SSAPRE operate on each lexically identified expression based only on its occurrence nodes. By applying the six steps of SSAPRE to each lexically identified expression individually, we can decouple PRE for each expression from the treatment of other expressions. Intermediate storage allocated for use in optimizing an expression can be recycled for use in optimizing the next expression. The total memory working set size needed to perform PRE on all the expressions in the program is thus substantially reduced. This scheme also allows parallel invocation of PRE for different lexically identified expressions where parallel processing facilities are available.

The occurrence nodes created by *Collect-Occurrences* are called *real* occurrence nodes, because they correspond to occurrences of the expression in the input program. There are other kinds of occurrence nodes represented during the six steps of SSAPRE. Based on the real occurrence nodes, Φ-*Insertion* creates Φ occurrence nodes to represent the Φ's that it inserts. From the Φ occurrence nodes, it also creates Φ-*predecessor* occurrence nodes, one at the end of each block that is a predecessor of some block containing a Φ. Φ-predecessor occurrences serve as place holders for Φ operands, as the operands are regarded as occurring at the predecessors of the block containing the Φ.

To represent the factored redundancy graph, each occurrence node has a *class* field for storing the redundancy class number assigned to it. For a Φ-predecessor node or a real occurrence node that represents a use (*i.e.*, one that is not the representative for its class), the *def* field points to the representative occurrence for the redundancy class; these fields represent upward edges in the factored redundancy graph. For Φ occurrence nodes, the Φ operands and result are provided.

Separately, there are *exit* occurrence nodes for indicating when we reach a point of program exit. They are used only in the *Rename* step for initializing the *down_safe* flag.

In the remaining steps of SSAPRE, we need to visit the occurrence nodes in an order corresponding to a preorder traversal of the dominator tree (DT) of the control flow graph, so we maintain the sequence of occurrence nodes in this sorted order. We precompute the *depth-first number* (*dfn*) and the number of descendents

(des) for each node in the DT. For any two basic blocks x and y, we can determine whether x dominates y using the formula:

$$Dominate(x,y) \equiv dfn(x) \leq dfn(y) \leq dfn(x) + des(x)$$

When we walk through the sequence of basic blocks in dominator-tree preorder, $Dominate(x,y) = true$ indicates that we are descending the DT. $Dominate(x,y) = false$ alerts us to the need to take appropriate action due to the fact that we are backtracking up the DT; in the case of *Rename*, it is necessary to pop the renaming stack until the version at the top of the stack is defined at a block that dominates y. These observations allow us to walk the occurrence list in dominator tree preorder without a recursive descent of the dominator tree.

5.2 Nested Expressions

Optimizing one expression at a time allows us to exploit the absence of redundancy in nested expression trees in speeding up SSAPRE. We use the following definition to explain what we mean:

DEFINITION 6. *A compound expression is an expression that consists of an operator that operates on the results of additional operators within the expression.*

For example, the expression $(a + b) - c$ is compound because it consists of the $-$ operator that operates on the result of $a + b$. In contrast, $a + b$ is a non-compound, or simple, expression. PRE needs to be applied to all the operations in a compound expression, because each of them may exhibit redundancy. Prior approaches to PRE based on bit vectors typically assign a separate bit vector slot to each operation in a compound expression and then apply PRE to all expressions encoded in the bit vectors simultaneously. However, we can take advantage of an important observation regarding redundancies in compound expressions:

OBSERVATION 2. *If redundancy exists in a compound expression, the same redundancy exists in all the operators within the expression. Conversely, if a simple expression does not exhibit any redundancy, no compound expression that contains the simple expression exhibits redundancy.*

For example, if redundancy exists for $(a + b) - c$, the same redundancy must exist for $a+b$. If $a+b$ does not exhibit redundancy, then $(a+b)-c$ also must not exhibit redundancy. If $a+b$ has redundancy, however, no inference can be drawn regarding redundancy for the $-$ operation in $(a + b) - c$.

As a consequence of Theorems 2 and 3, elimination of redundancy always results in converting the expression to the use of a temporary, so the above observation leads to a strategy for dealing with the optimization of compound expressions. The strategy is to defer PRE for compound expressions until they become converted to simple expressions by redundancy elimination for their constituent expressions. In our worklist-driven approach, this implies that only simple expressions are allowed in the worklist. As their optimizations proceed, some simple expressions will be converted to temporaries, which in turn causes some compound expressions to become simple expressions. As new simple expressions are formed, they are entered into the worklist.

As an example, for $(a+b)-c$, $a+b$ is a simple expression, and is entered into the worklist by *Collect-Occurrences*. After SSAPRE has worked on $a+b$, any redundant

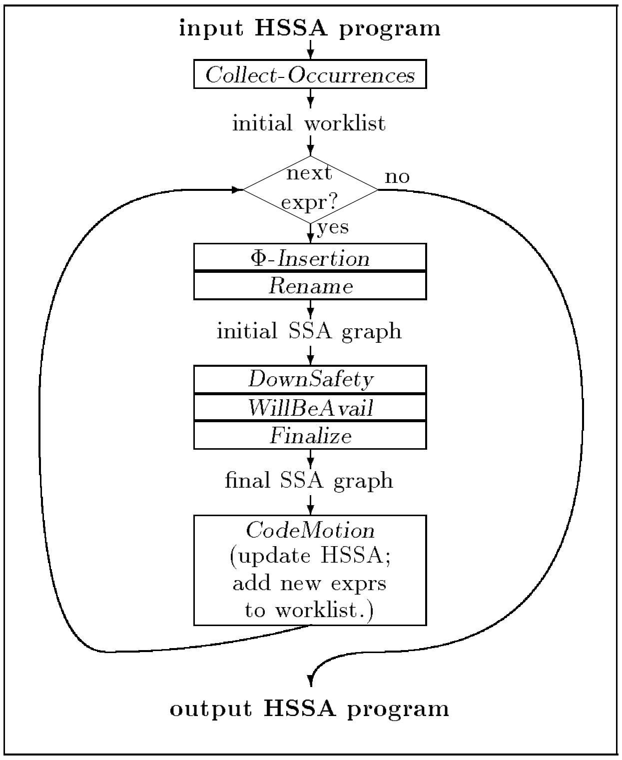


Fig. 17. SSAPRE implementation flow chart

occurrence of $a + b$ will be replaced by a temporary t. If PRE on $a + b$ converts $(a + b) - c$ to $t - c$, this new simple expression, formed in the *CodeMotion* step, will be entered as a new member of the worklist. Redundancies of $t - c$, and hence redundancies in $(a + b) - c$, will be eliminated later when SSAPRE processes $t - c$. If the expression $(a + b) - c$ does not yield $t - c$ when $a + b$ is processed, $(a + b) - c$ will remain a compound expression, and will never be processed by SSAPRE.

In the absence of redundancy, SSAPRE terminates quickly because it can skip the processing of all compound expressions. In the presence of redundancies, the approach has the secondary effect of converting the evaluation of compound expressions essentially to triplet form, because the result of each simple expression is saved to a temporary, which is then used as an operand in the evaluation of another simple expression. If this effect is undesirable, the compound expressions can be reconstructed by performing copy propagation on the temporaries that have only one locally-occurring use. After copy propagation, the temporaries can be eliminated by dead store elimination. But in the usual case the program will eventually be translated to machine instructions, so the triplet form poses no obstacle for most target architectures.

The above strategy deals cleanly with the interaction between the optimizations of nested expressions, while speeding up optimization by skipping compound expressions that exhibit no redundancy. This strategy is hard to implement in bit-vector

PRE, which typically works on all expressions in the program simultaneously so as to take advantage of the parallelism possible with bit-vector operations. And because SSAPRE only has to deal with simple expressions, its implementation can be simplified. Figure 17 shows the flow chart of an implementation of SSAPRE based on worklists that incorporates the above strategy of dealing with compound expressions.

5.3 Demand-driven Φ Insertion

The Φ-*Insertion* algorithm given in Section 3.1 is not sparse because it can insert many Φ's due to variable assignments that do not alter any occurrence of the corresponding expression. In particular, we only need to insert a Φ at a merge point when it reaches a later occurrence of the expression (*i.e.*, when the expression is partially anticipated at the merge point), because otherwise the Φ will not contribute to any optimization in PRE and need not correspond to a ϕ in the final SSA form for the expression's real temporary. In this section, we present a technique that substantially reduces the number of unnecessary Φ's inserted. The resulting algorithm is sparse in the sense that all the Φ's inserted are justified either by appearing on the iterated dominance frontier of some real occurrence of the expression or by appearing at a point where the expression is partially anticipated.

Recall from Section 3.1 that Φ's are placed on the iterated dominance frontiers of real occurrences of the expression and of assignments to operands of the expression, because these points necessarily contain the combined iterated dominance frontiers of the set of assignments to the real expression temporary. In our sparse Φ insertion algorithm, both types of Φ insertions are performed together in one pass over the program, with the second type of Φ insertion performed in a demand-driven way. We use the set *DF_phis*$[i]$ to keep track of the Φ's inserted on the iterated dominance frontiers of the occurrences of expression $E^{(i)}$. We use the set *Var_phis*$[i][j]$ to keep track of the Φ's inserted due to the occurrence of ϕ's for the j^{th} variable in expression $E^{(i)}$. When we come across an occurrence of expression $E^{(i)}$, we update *DF_phis*$[i]$. For each variable v_j in the occurrence, we check if it is defined by a ϕ. If it is, we update *Var_phis*$[i][j]$, because a Φ at the block that contains the ϕ for v_j may participate in optimization of the current occurrence of $E^{(i)}$. The same may apply to earlier points in the program as well, so it is necessary to check recursively for updates to *Var_phis*$[i][j]$ for each operand in the ϕ for v_j. After all occurrences in the program have been processed, the places to insert Φ's for $E^{(i)}$ are given by the union of *DF_phis*$[i]$ with the *Var_phis*$[i][j]$'s. The full algorithm for the Φ-*Insertion* step is given in Figure 18. Using this demand-driven technique, we take advantage of the SSA representation of the input program.

The following lemma replaces Lemma 1 in the demand-driven context.

LEMMA 10. *(Sufficiency of Φ insertion) If B is a basic block where no Φ is inserted and the expression is partially anticipated at the entry to B, exactly one evaluation of the expression can reach the entry to B.*

PROOF. Suppose two different evaluations of the expression, ψ_1 and ψ_2, reach the entry to B. It cannot be the case that ψ_1 and ψ_2 both dominate B; suppose without loss of generality that ψ_1 does not dominate B. Now there exists a block B_0 that dominates B, is reached by ψ_1 and ψ_2, and lies in $\mathrm{DF}^+(\psi_1)$ (*n.b.*, B_0 may

```
procedure Set_var_phis(phi, i, j)
  if (phi ∉ Var_phis[i][j]) {
    Var_phis[i][j] ← Var_phis[i][j] ∪ {block containing phi}
    for each operand V of phi do
      if (V is defined by φ)
        Set_var_phis(Phi(V), i, j)
  }
end Set_var_phis

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 j in E^(i) do {
      let V be the SSA variable in the j^th position in X
      if (V is defined by φ)
        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
```

Fig. 18. Algorithm for demand-driven Φ insertion

be B). If ψ_1 is a real occurrence or a Φ, the Φ-*Insertion* step must have placed a Φ in B_0, contradicting the proposition that ψ_1 reaches B. If on the other hand ψ_1 is an assignment to an operand ν of the expression (so $\perp$ is among the values reaching B), there must be a ϕ for ν in B_0 by the correctness of the input SSA form. Hence when Φ-*Insertion* processed the expression occurrence responsible for the partial anticipation at B, it must have placed a Φ in B_0, once again contradicting the proposition that ψ_1 reaches B. $\square$

5.4 Delayed Renaming

The *Rename* algorithm described in Section 3.2 maintains version stacks for all the variables in the program in addition to the redundancy class stacks for the expressions. Apart from taking up additional storage, updating the variable version stacks requires keeping track of changes to the values of the expressions' variables. Since many versions of variables may not appear in any PRE candidate expression, the algorithm is not sparse. One goal of the worklist-driven approach is to be able to perform the six steps of SSAPRE based solely on the occurrence nodes, so we desire a *Rename* algorithm that can perform its job without passing over the entire program. We now describe the *delayed renaming* technique, which is a more efficient version of the *Rename* step of SSAPRE.

	defining top-of-stack occurrence X	current occurrence Y	condition for assigning class number of X to Y	applying phase
case 1	real	real	all corresponding variables	*Rename1*
case 2	real	Φ operand	have same SSA versions	*Rename2*
case 3	Φ	real	definitions of all	*Rename1*
case 4	Φ	Φ operand	variables in Y dominate X	*Rename2*

Table I. Assigning class numbers in *Rename*

We begin by discussing in greater detail how redundancy class numbers are assigned by the method given in Section 3.2. The *Rename* step maintains the redundancy class stack so that, at any time, the top of the stack gives the last class number, with the corresponding defining occurrence node, assigned to the expression in the preorder traversal of the DT. Every Φ defines a new class, so the question of whether to assign new class numbers applies only to real occurrences and Φ operands. This, plus the fact that the defining occurrence on the top of the stack must correspond to either a real occurrence or a Φ, leads to only four different situations that can arise; these situations are shown in Table I. A real occurrence of the expression or a Φ operand is given the class number of the top of the expression renaming stack if the versions of all the variables in that occurrence match the current versions given by the renaming stacks for the variables. This decision on the question of whether to assign a new class number is the sole purpose of the variable stacks in the *Rename* algorithm of Section 3.2.

If the defining occurrence at the top of the expression renaming stack is a Φ, the versions of the variables at the Φ are not provided by the Φ occurrence node. This situation corresponds to cases 3 and 4 in Table I. In these cases, *Rename* uses a different method for determining whether the current version of a variable matches the version of the same variable in that last Φ occurrence of the expression. Let A be the beginning of the basic block containing the Φ, and B be the location of the current occurrence of the expression. Suppose we are considering variable x in the expression. Let C be the assignment that defines the current version of x. By the definition of SSA, C dominates B. Using $\gg$ to denote the dominance relation, $C \gg B$. Because of our maintenance of the expression renaming stack in the preorder traversal of the DT, $A \gg B$. Thus, given $C \gg B$ and $A \gg B$, either $A \gg C$, or $C \gg A$. $A \gg C$ implies the version of x at A is different from the version at B. $C \gg A$ implies the version of x at A is the same as the version at B. Thus, $C \gg A$ is a necessary and sufficient condition for the version of x to be the same at both A and B. This is the condition we use for cases 3 and 4 in Table I. (If A and C are respectively a Φ and a ϕ in the same basic block, we say C dominates A).

The variable stacks are unnecessary for cases 1 and 3 because the variable versions are given explicitly by the expressions themselves and we do not have to rely on the variable stacks to find the current versions of the variables in the expression. However, for cases 2 and 4, we have difficulty in renaming the Φ operand, because no real occurrence of the expression is available to provide the the current versions of the variables. We solve this problem using the delayed renaming strategy, in which we split the *Rename* step into two separate passes. The first pass, *Rename1*, is *Rename* minus the use and maintenance of variable stacks. When renaming

```
procedure Assign_new_class(occ)
  class(occ) ←count
  Push(occ, stack)
  count ← count + 1
end Assign_new_class

procedure Rename1
  count ← 0
  stack ← empty
  set_for_rename2 ← {}
  for each occurrence Y of the current expression in preorder DT traversal order do {
    while (Top(stack) does not dominate Y) do
      Pop(stack)
    if (Y is a Φ occurrence)
      Assign_new_class(Y)
    else if (Y is a real occurrence)
      if (stack is empty)
        Assign_new_class(Y)
      else {
        X ← Top(stack)
        if (X is a real occurrence)
          if (all corresponding variables in Y and X have same SSA version)
            class(Y) ← class(X)
            def(Y) ← X
          else Assign_new_class(Y)
        else /* X is a Φ occurrence */
          if (definitions of all variables in Y dominate X) {
            class(Y) ← class(X)
            def(Y) ← X
            set_for_rename2 ← set_for_rename2 ∪ {Y}
          }
          else Assign_new_class(Y)
      }
    else if (Y is a Φ operand occurrence)
      if (stack is empty)
        def(Y) ← ⊥
      else {
        X ← Top(stack)
        class(Y) ← class(X)
        def(Y) ← X
      }
  }
end Rename1
```

Fig. 19. Algorithm for *Rename1*

a Φ operand (cases 2 and 4), instead of applying the method shown in Table I, *Rename1* optimistically assumes its version to be the version given by the top of the expression's version stack. Note that either this assumption is correct or the operand should be $\perp$. No other class number can be right. *Rename1* performs all its work based solely on the occurrence nodes of the expression and the expression's version stack that it maintains while visiting the occurrence nodes. Whenever *Rename1* encounters a real occurrence of the expression that is defined by Φ, it adds the real occurrence to a set that it builds for the second pass to use. The

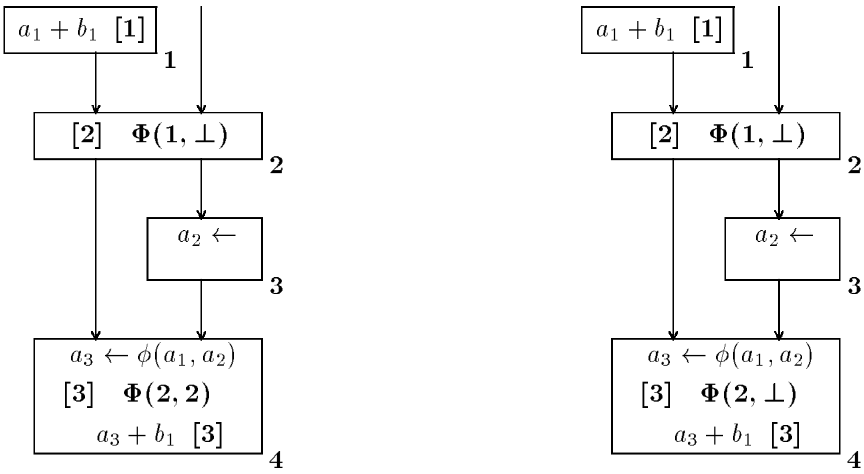
```
function φ_opnd_from_res(Z, j)
   b ← block containing Φ that defines Z
   Q ← copy of Z
   for each variable v in Z do {
      if (v is defined by φ in b)
         replace v in Q by j^th operand of v's φ
   }
   return Q
end φ_opnd_from_res

procedure Rename2
   for each f ∈ F do
      for each operand ω of f do
         processed(ω) ← false
   while (set_for_rename2 is not empty) do {
      remove real occurrence Z from set_for_rename2
      f ← Φ that defines Z
      for each operand ω of f do {
         if (not processed(ω)) {
            j ← index of ω in f
            Y ← φ_opnd_from_res(Z, j)
            X ← def(ω) (as assigned by Rename1)
            if (X is a real occurrence)
               if (all corresponding variables in Y and X have same SSA version)
                  /* no change needed */
               else
                  def(ω) ← ⊥
            else /* X is a Φ occurrence */
               if (definitions of all variables in Y dominate X)
                  /* no change needed */
                  set_for_rename2 ← set_for_rename2 ∪ {Y}
               else
                  def(ω) ← ⊥
            processed(ω) ← true
         }
      }
   }
end Rename2
```

Fig. 20. Algorithm for *Rename2*

algorithm for *Rename1* is given in Figure 19.

The graph built by *Rename1* is optimistic in the sense that it presumes more redundancy than may actually be present. The final renaming of Φ operands is delayed to the second pass, *Rename2*, which is given in Figure 20. *Rename2* works according to the set built for it by *Rename1* that contains all the real occurrences defined by Φ's. Each such real occurrence provides the current versions of the variables at the Φ. From the version of each variable at the Φ, *Rename2* determines the version of the variable at each predecessor block based on the presence or absence of ϕ for the variable at that merge block (function ϕ*_opnd_from_res*). Then the algorithm applies the methods for cases 2 and 4 of Table I, except that the most recent defining occurrence is retrieved through an upward edge of the FRG rather than from the top of the expression renaming stack. If the Φ operand assigned by



(a) after *Rename1* (b) after *Rename2*

Fig. 21. Operation of delayed renaming

Rename1 is not correct, *Rename2* resets it to $\perp$. If the Φ operand is correct and is defined by another Φ, *Rename2* manufactures a real occurrence node containing the versions of the variables at the Φ operand and adds the manufactured occurrence to the set to recursively ensure verification of the variable versions for operands of the defining Φ.

In the example of Figure 21, *Rename1* sets both operands of the Φ in block 4 to refer to class 2 because the Φ representing class 2 in block 2 appears at the top of the expression renaming stack when those operands are encountered. *Rename1* also places block 4's real occurrence into the set of occurrences for processing by *Rename2* because class 3 is represented by a Φ. When *Rename2* processes that real occurrence, the algorithm discovers that class 2 is correct as the first operand of the Φ representing class 3. This determination is made as follows: The current versions of the expression's variables at the end of block 2 where the Φ operand occurs are found to be a_1 (because it is the first operand of the ϕ for a in block 4) and b_1 (because it is the version that appears in the real occurrence, and there is no ϕ for b in block 4). The algorithm concludes that class 2 is correct as the first Φ operand because the definitions of a_1 and b_1 both dominate the representative occurrence of class 2. Upon concluding this operand is correct, the algorithm builds the expression $a_1 + b_1$ viewed as occurring at the end of block 2, and enters this occurrence in the set of occurrences to be processed. For the second Φ operand in block 4, the algorithm discovers that the definition of a_2 (the current version at the end of block 3) does not dominate the Φ that represents class 2. Therefore the second operand of block 4's Φ is reset to $\perp$.

Delayed renaming relies on seeing a later real occurrence of the expression to determine the versions of the variables at the Φ and thus the Φ operands. A later real occurrence will be seen only if the expression is partially anticipated at the Φ.

It is sometimes more efficient to eliminate dead Φ's early in the process of building the FRG (see Section 5.5). Any Φ where the expression is not partially anticipated is guaranteed to be dead in the final SSA form for the expression temporary, so the delayed renaming algorithm incorporates the additional function of determining Φ's that are not live. Without delayed renaming, this task would require a dead store elimination pass.

5.5 Efficient *save* computation in *Finalize*

Recall from Section 3.5 that in order to compute the *save* predicate for each real occurrence, the *Finalize* algorithm searches along upward edges in the FRG from real occurrences satisfying *reload* and sets *save* for each real occurrence it encounters that is an available definition. There are two ways an implementor might reduce the cost of this *save* computation.

The first technique is based on the observation that any real occurrence that is the available definition for a real occurrence or for an operand of a Φ satisfying $will\_be\_avail \wedge down\_safe$ must be saved to the temporary. Such real occurrences can be recognized during the processing of Φ operands in the first part of *Finalize*, and their *save* predicates can be set at that time. The remaining *save* predicates must then be established through the graph search in the second part of *Finalize*, but this search can be restricted to those Φ's that satisfy $will\_be\_avail \wedge \neg down\_safe$. If demand-driven Φ insertion and delayed renaming have been used, the number of such Φ's is likely to be quite small, so the benefit of this approach for compilation time may be noticeable.

The second technique that can be used in practice to set the *save* predicate relies on the observation that with most target architectures, generating intermediate code that saves an expression result to a temporary will cost nothing because expression results must be computed in registers at the level of machine code anyway. Consequently, an implementor might feel it worthwhile to dispense entirely with the graph search to determine the *save* predicate, and replace it with a simple heuristic that sets *save* for every real occurrence that is the available definition for another real occurrence or for an operand of a Φ that satisfies *will_be_avail*. This heuristic clearly sets *save* for every occurrence that must be saved, and may set *save* for some others. The main disadvantage of the heuristic is that setting *save* unnecessarily can make the final SSA form for the expression temporary non-minimal.

6. ANALYSIS

While the formulation of the optimal code motion algorithm in SSAPRE is self-contained, we can gain additional insight by comparing SSAPRE with a slotwise implementation of lazy code motion. We can regard the Φ-*Insertion* and *Rename* steps to construct the factored redundancy graph as corresponding to the initialization of data flow information; these two steps are faster in SSAPRE because we take full advantage of the SSA form of the input program. While down-safety corresponds to the same attribute in lazy code motion, the correlation in the part that involves forward propagation of data flow information is less direct. Since we have shown that our algorithm yields the same results as lazy code motion, it is quite plausible that the forward propagation parts in SSAPRE and a slotwise implementation of lazy code motion can be proven essentially equivalent. But because slotwise analysis propagates with respect to the control flow graph and SSAPRE propagates with respect to the sparse SSA graph, the propagation in SSAPRE will take fewer steps. This effect is heightened by the tendency of the *DownSafety*, *CanBeAvail*, *Later*, *Set_save*, and *Set_replacement* searches to limit the sections of the graph that must be considered by subsequent steps. The factored redundancy graph also allows SSAPRE to maintain the generated temporary easily in SSA form.

The complexities of the various steps in SSAPRE can be easily established. Assuming the implementation described in Section 5, the *Rename, DownSafety, WillBeAvail, Finalize* and *CodeMotion* steps are all linear with respect to the sum of the number of nodes (v) and edges (e) in the FRG. The Φ*-Insertion* step is $\Omega(v^2)$ for insertion at domination frontiers, but as we explained in Section 3.1, there are linear-time SSA ϕ-placement algorithms that can be used to lower it to $O(e)$. The second kind of Φ insertion due to variable ϕ's is also linear using our demand-driven algorithm. Thus, for a program of size n, SSAPRE's total time is $O(n(E+V))$, where E and V are the number of edges and nodes in the control flow graph respectively. This is pleasing given that SSAPRE replaces both the solution of data flow equations and the initialization of the local data flow attributes in bit-vector-based PRE.

7. MEASUREMENTS

In this section, we repeat the compile-time performance measurements for the SPECint95 and SPECfp95 benchmark suites from [Chow et al. 1997] and the related discussion contrasting the compilation efficiencies between a bit-vector-based implementation of PRE and an implementation of SSAPRE. Then we offer another perspective on the compilation efficiency of SSAPRE by presenting statistical data that characterize the partial redundancy problems in the same two benchmark suites. Our statistics are generated using the optimizer WOPT, a component of the Silicon Graphics MIPSpro Compiler Suite. WOPT is an intraprocedural global optimizer that uses SSA as its internal program representation for performing all its optimizations [Liu et al. 1996; Chow et al. 1996; Kennedy et al. 1998; Lo et al. 1998]. The SSAPRE phase in WOPT incorporates the practical implementation techniques described in Section 5.

For our measurements the benchmarks were compiled at the optimization level `-O2`, in which only intraprocedural analyses and optimizations are performed. Our implementation of SSAPRE incorporates the additional functionalities of strength reduction and linear function test replacement, as described in [Kennedy et al. 1998]. We have suppressed these extra optimizations in collecting the statistics so that the results only reflect the effects of partial redundancy elimination.

In the remainder of this section, we present and discuss four sets of statistical data. In Section 7.1, we compare the time spent in performing PRE between a bit-vector-based implementation and our implementation of SSAPRE. In Section 7.2, we measure the fraction of expressions that require full processing by SSAPRE. In Section 7.3, we estimate the degree of sparseness that can be achieved in SSAPRE by measuring the size of the FRG divided by the size of the control flow graph. In Section 7.4, we provide statistics on the PRE problems in the benchmarks and the results of applying PRE to them.

Benchmark	Bit-vector PRE (T1)	SSAPRE (T2)	Ratio T2/T1
go	116900	151260	1.293
m88ksim	4850	4440	0.915
gcc	886360	339160	0.382
compress	100	60	0.600
li	12950	5090	0.393
ijpeg	10340	11200	1.083
perl	98840	34970	0.353
vortex	62950	53000	0.841

Table II. Time (in msec.) spent in PRE in compiling SPECint95

Benchmark	Bit-vector PRE (T1)	SSAPRE (T2)	Ratio T2/T1
tomcatv	40	60	1.500
swim	170	400	2.352
su2cor	500	700	1.399
hydro2d	7080	8780	1.240
mgrid	500	1400	2.799
applu	5060	9450	1.867
turb3d	2420	5000	2.066
apsi	37930	93960	2.477
fpppp	1450	1980	1.365
wave5	94150	85800	0.911

Table III. Time (in msec.) spent in PRE in compiling SPECfp95

7.1 Optimization Time Measurements

The WOPT optimizer uses a variant of SSA called HSSA as its internal program representation [Chow et al. 1996]. Prior to our SSAPRE implementation in the MIPSpro version 7.2 compilers, the optimizer had used the bit-vector-based Morel and Renvoise algorithm [Chow 1983] to perform PRE, while it used known SSA-based algorithms for its other optimizations. In this section, we compare the performance of SSAPRE and the bit-vector-based implementation using the SPECint95 and SPECfp95 benchmark suites.

Measured by the running time of the optimized benchmark code, the differences between the two implementations of PRE are not noticeable. We are more interested in comparing the optimization efficiencies between the sparse approach and the bit-vector approach. Both implementations of PRE begin with an SSA representation of the program. The bit-vector-based PRE starts by determining the local attributes and setting up the bit vectors for data flow analyses. Our bit vectors are represented as arrays of 64-bit words, and their operations are very efficient. The bit-vector-based PRE does not update the SSA representation of the program; instead it encodes the effects of PRE in bit vector form until it is ready to emit the output program in a non-SSA representation. Our timing for the bit-vector-based PRE includes only the local attributes phase and the solution time of the PRE data flow equations. Correspondingly, we omit the *CodeMotion* step from the SSAPRE timing and include only the *Collect-Occurrences* pass and the first five SSAPRE steps. Tables II and III give our timing results as measured on a 195 MHz R10000 Silicon Graphics Power Challenge.

The measurements in Tables II and III show widely different results across the various benchmarks. In the SPECint95 benchmarks, SSAPRE uses from 65% less time than the bit-vector approach in `perl` to 29% more in `go`. In the SPECfp95 benchmarks, SSAPRE is usually slower, sometimes by up to 2.8 times, as in the case of `mgrid`. Without examining the sizes and characteristics of each benchmark's procedures in detail, we cannot characterize from these measurement results the situations in which our SSAPRE implementation is superior to our bit-vector implementation. Even so, we see that the efficiency of sparse implementation stands out mainly in large procedures. In small procedures, a sparse graph cannot be much simpler than the control flow graph, so it is much harder to beat the performance of bit vectors that process 64 expressions at a time. The advantage of sparse implementations increases with procedure size. In large procedures, many expressions do not appear throughout the procedure, and their sparse representations are much smaller compared to the control flow graph.

Despite the strong bias towards bit-vector-based PRE being faster in our set of measurements, we think SSAPRE is very promising. The time complexity of collecting local attributes is $\Omega(n^3)$. A number of techniques contribute to speeding up bit-vector data flow analysis, but there is little promise of overcoming the cubic complexity of local attribute collection in the bit-vector approach. As data flow analyses have sped up, the time spent collecting local attributes has come to dominate: our bit-vector-based PRE spends 51% of its time in its local attributes collection phase while optimizing our benchmarks. Because of the cubic complexity, optimization efficiency is more of an issue in large procedures. With the trend towards more inlining during compilation, large procedures will be more commonplace, and the efficiency advantages of sparse implementation will become more obvious.

7.2 Expression Candidates

In Section 5.2, we show an implementation scheme in which SSAPRE avoids working on a compound expression until PRE has converted it to a simple expression. The scheme is based on the observation that if a simple expression exhibits no redundancy, any compound expression that contains it also exhibits no redundancy. The scheme also relies on the fact that if a compound expression has redundancy, its component expressions will eventually be converted to temporaries, causing the expression to become a simple expression.

In Tables IV and V, column A gives the number of program units in each benchmark. For each benchmark, we count the number of lexically identified expressions in each program unit, and sum them across all the program units in the benchmark. The total, which represents the number of PRE problems in each benchmark, is shown in column B. Column C shows the number of expressions which are simple expressions or become converted to simple expressions at the end of SSAPRE; it represents the number of lexically identified expressions that SSAPRE has to work on. Column C/B shows that, under our scheme for exploiting the absence of redundancy in nested expression trees, SSAPRE only has to process between 46% to 65% of the PRE candidates that traditional PRE schemes have to handle.

benchmark	A program units	B total exprs	C simple exprs	C/B	D exprs not bypassed	D/B
go	372	20094	9398	47%	8296	41%
m88ksim	252	6501	3534	54%	2722	42%
gcc	1997	65607	30083	46%	25334	39%
compress	24	382	245	64%	199	52%
li	357	1878	994	53%	666	35%
ijpeg	466	11541	6526	57%	5251	45%
perl	273	8881	4357	49%	3621	41%
vortex	923	14284	7414	52%	6683	47%
average				53%		43%

Table IV. Lexically identified expressions in SPECint95

benchmark	A program units	B total exprs	C simple exprs	C/B	D exprs not bypassed	D/B
tomcatv	1	177	103	58%	98	55%
swim	6	559	337	60%	305	55%
su2cor	26	2336	1418	61%	1328	57%
hydro2d	42	1904	1093	57%	976	51%
mgrid	12	808	502	62%	482	60%
applu	16	3911	2024	52%	1972	50%
turb3d	23	2342	1317	56%	1123	48%
apsi	96	7109	4617	65%	4249	60%
fpppp	38	5673	3326	59%	2251	40%
wave5	93	7444	4685	63%	4370	59%
average				59%		53%

Table V. Lexically identified expressions in SPECfp95

Another way to take advantage of the per-expression processing mode in speeding up PRE is to detect when the problem has a trivial answer. We find that many PRE candidates only occur once in the entire program unit; if that occurrence does not result in the insertion of any Φ, we can conclude that there is no PRE opportunity for that expression. Thus, at the end of the Φ-*Insertion* step, we check if there is any Φ inserted; if none is inserted, and the expression only occurs once in the program unit, we bypass the rest of the SSAPRE steps. This method allows the number of PRE candidates that require full processing by SSAPRE to be further reduced. Column D in Tables IV and V shows the total number of expressions in each benchmark that require full processing by SSAPRE after application of the above check for bypassing. Column D/B shows that by combining the above two schemes, SSAPRE has to fully process less than half of the original PRE candidates. The rest of our measurements are made only on these fully processed expression candidates.

7.3 Density

We define *density* to be the quotient of the size of the SSA graph formed during SSAPRE and the size of the control flow graph of the program unit. A low value

benchmark	average over each benchmark A nodes	B edges	C density
go	7.6	10.7	0.13
m88ksim	5.6	7.3	0.15
gcc	13.4	24.8	0.13
compress	4.4	4.6	0.22
li	5.0	6.5	0.22
ijpeg	6.5	8.0	0.26
perl	14.6	27.6	0.15
vortex	10.6	15.2	0.20
average			0.18

Table VI. Density of PRE problems in SPECint95

benchmark	average over each benchmark A nodes	B edges	C density
tomcatv	14.3	20.8	0.15
swim	6.2	6.7	0.24
su2cor	11.0	13.4	0.20
hydro2d	8.4	10.3	0.23
mgrid	7.5	9.0	0.32
applu	13.8	18.7	0.36
turb3d	10.1	11.5	0.48
apsi	7.7	9.1	0.25
fpppp	19.3	36.0	1.93
wave5	10.4	12.3	0.17
average			0.43

Table VII. Density of PRE problems in SPECfp95

for density implies that the FRG is much simpler than the control flow graph, so that the sparse approach has greater speed advantage compared to any solution method based on the control flow graph. We compute the size of a graph as the number of nodes plus the number of edges. For our factored redundancy graphs, the nodes are either real occurrences or Φ's. The number of edges in the FRG is equal to the number of real occurrences that reuse an existing class number (*i.e.*, that are not assigned a new class number), plus the number of Φ operands in the entire SSA graph. We perform this measurement during *Rename1* of the delayed renaming algorithm, when the FRG representation is at its largest.

For each PRE candidate, we perform the above measurements and compute the density of its FRG. Then we average the data over all the PRE candidates in each benchmark. Tables VI and VII show the results for the SPECint95 and SPECfp95 benchmarks respectively. Columns A and B give the average number of nodes and edges respectively in the factored redundancy graphs for the PRE candidates in each benchmark. Column C gives the average density of the FRGs in each benchmark. The average density ranges between 0.13 and 0.48, with the exception of `fpppp`, which shows a density of 1.93. `fpppp`'s density is skewed by the routine `FPPPP`; the average density for the routine `FPPPP` alone is 7.4, because it has only a single basic

benchmark	A real occs	B Φ's	B/A	C insertions	C/A	D deletions	D/A
go	2.9	4.7	164%	0.11	3.9%	1.47	51%
m88ksim	2.4	3.2	137%	0.16	6.9%	0.88	37%
gcc	2.7	10.7	396%	0.13	4.7%	1.00	37%
compress	1.7	2.6	150%	0.07	3.7%	0.56	32%
li	1.6	3.4	214%	0.04	2.7%	0.18	11%
ijpeg	2.3	4.1	176%	0.10	4.2%	0.66	28%
perl	2.8	11.8	424%	0.12	4.2%	0.69	25%
vortex	2.3	8.2	356%	0.28	11.9%	0.72	31%
average	2.3	6.1	252%	0.13	5.3%	0.77	32%

Table VIII. Average factored redundancy graphs in SPECint95

benchmark	A real occs	B Φ's	B/A	C insertions	C/A	D deletions	D/A
tomcatv	4.1	10.2	250%	0.13	3.2%	1.96	48%
swim	2.9	3.3	116%	0.18	6.3%	1.58	55%
su2cor	4.8	6.2	130%	0.18	3.7%	2.69	56%
hydro2d	3.3	5.1	155%	0.31	9.4%	1.47	45%
mgrid	2.8	4.7	170%	0.31	11.2%	1.32	47%
applu	4.9	8.8	179%	0.30	6.1%	3.08	63%
turb3d	5.2	5.0	97%	0.17	3.4%	3.14	61%
apsi	3.0	4.7	155%	0.26	8.6%	1.56	52%
fpppp	5.6	13.7	244%	0.08	1.4%	3.88	69%
wave5	4.3	6.1	142%	0.25	5.7%	2.21	51%
average	4.1	6.8	164%	0.22	5.9%	2.29	55%

Table IX. Average factored redundancy graphs in SPECfp95

block that contains hundreds of expression occurrences.

Among the SPECint95 benchmarks, the average density is 0.18. Among the SPECfp95 benchmarks but excluding `fpppp`, the average density is 0.27. The lower density in the SPECint95 benchmarks accounts for the observation in Section 7.1 that SSAPRE's compile-time performance relative to a bit-vector-based PRE implementation is better for SPECint95 than for SPECfp95.

7.4 PRE Opportunities

We characterize PRE problems by counting the number of real occurrences and Φ's in the FRG. Opportunities for PRE are represented by the number of insertions and deletions performed. We perform these measurements for each FRG and average them over all the PRE candidates in each benchmark. Columns A, B, C, and D in Tables VIII and IX show these data for the SPECint95 and SPECfp95 benchmarks, respectively. The additional columns show the ratios (in percent) of Φ's, insertions and deletions to the real occurrences.

The factored redundancy graphs in the SPECfp95 benchmarks have more real occurrences than those in the SPECint95 benchmarks, though the two benchmark suites generate similar numbers of Φ's per expression. There are more deletions in the SPECfp95 benchmarks than in the SPECint95 benchmarks: on average, 55% of the real occurrences are deleted in SPECfp95, while only 32% are deleted in

SPECint95. In contrast, insertions are quite rare, with only one insertion in every five factored redundancy graphs in the SPECfp95 benchmarks. This shows that the majority of the deletions are due to full rather than partial redundancy.

`li` shows the exceptionally low deletion percentage of 11%. According to Table IV, `li` has 1878 lexically identified expressions distributed over 357 program units. This translates to 5 expressions per program unit, out of which only 2 expressions require full processing by SSAPRE. This low density of expressions requiring full processing is atypical and may be related to the low incidence of redundancy in `li`.

Overall, the number of deletions shown in Tables VIII and IX confirms the importance of PRE in optimizing compilers.

8. CONCLUSION

In this paper we present a sparse approach to the problem of redundancy elimination based on a factored representation of redundancy relations for expressions in the program. Factoring at relevant control flow merge points is essential to exposing partial redundancies, and this observation highlights for the first time the close relationship between PRE and SSA form. The data flow analyses of PRE are all focused on the locations of the factoring operator Φ. The SSAPRE framework capitalizes on several prior techniques for computing and manipulating SSA form. Meanwhile, SSAPRE depends on its input program being in SSA form for maximum efficiency, and intrinsically produces its output in SSA form. SSAPRE thus enables PRE to be seamlessly integrated into a global optimizer that uses SSA as its internal representation. We have implemented SSAPRE as the redundancy elimination framework in the MIPSpro version 7.2 compilers, and have gained valuable practical experience and empirical insight into the redundancy characteristics of a broad cross-section of real programs.

Previous uses of SSA were directed at problems related to variables. SSAPRE represents the first use of SSA to solve data flow problems related to expressions in the program. This work opens up the possibility to solve other expression-based data flow problems by representing them in the form of factored dependency edges and performing data flow analyses on the resulting sparse graph. In [Lo et al. 1998], we have applied this approach in performing load and store placement optimizations. Other candidate optimizations for using this framework are code hoisting, register shrink-wrapping [Chow 1988] and live range shrinking.

PRE has traditionally provided the context for integrating additional optimizations into its framework; one such optimization is operator strength reduction [Joshi and Dhamdhere 1982; Chow 1983; Dhamdhere 1989; Knoop et al. 1993; Dhaneshwar and Dhamdhere 1995]. In [Kennedy et al. 1998], we present techniques that allow strength reduction and linear function test replacement to be performed in the SSAPRE framework. In [Lo et al. 1998], we present techniques to incorporate speculative code motion in the SSAPRE framework, including one that can use execution profile data to improve code placement over what can be accomplished by PRE without speculation. Combining optimizations permits synergy among their effects, with results that often exceed expectations.

A. NOTATION AND CONVENTIONS

In this section we offer a table of symbols with a terse definition of each one and a pointer for each to the section of our paper where the item is defined or explained in more detail.

notation	English term	for more detail, see
DT	dominator tree	Section 2.1, page 4.
DF	dominance frontier	Section 2.1, page 4.
DF^+	iterated dominance frontier	Section 2.1, page 4.
ϕ	SSA factoring operator	Section 2.2, page 4
v_i	SSA version i of variable v	Section 2.2
Φ	redundancy factoring operator	Section 2.3, page 7
$\perp$	non-partially-redundant operand of Φ	Section 2.3, page 7
E	an arbitrary computation or expression	Section 2.3
E_i	particular occurrence of a computation	Section 2.3
$E^{(j)}$	j^{th} lexically identified expression	Section 3, page 11
t	expression temporary	Section 2.5
ψ_i	expression evaluation	Definition 5, page 13
$\mathcal{F}$	set of Φ's for current expression	Section 3, page 12
ν	an arbitrary expression operand	Lemmas 1, 4, and 10

ACKNOWLEDGEMENTS

The authors would like to thank Rune Dahl and the anonymous referees; their comments on an earlier draft are responsible for substantial improvments in our presentation.

REFERENCES

ALPERN, B., WEGMAN, M. N., AND ZADECK, F. K. 1988. Detecting equality of values in programs. In *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages*. 1–11.

BRIGGS, P. AND COOPER, K. 1994. Effective partial redundancy elimination. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*. 159–170.

BRIGGS, P., COOPER, K. D., AND SIMPSON, L. T. 1997. Value numbering. *Software Practice and Experience 27,* 6 (June), 701–724.

CHOI, J., CYTRON, R., AND FERRANTE, J. 1991. Automatic construction of sparse data flow evaluation graphs. In *Conference Record of the Eighteenth ACM Symposium on Principles of Programming Languages*. 55–66.

CHOI, J., SARKAR, V., AND SCHONBERG, E. 1996. Incremental computation of static single assignment form. In *Proceedings of the Sixth International Conference on Compiler Construction*. 223–237.

CHOW, F. 1983. A portable machine-independent global optimizer – design and measurements. Tech. Rep. 83-254 (PhD Thesis), Computer Systems Laboratory, Stanford University. Dec.

CHOW, F. 1988. Minimizing register usage penalty at procedure calls. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*. 85–94.

CHOW, F., CHAN, S., KENNEDY, R., LIU, S., LO, R., AND TU, P. 1997. A new algorithm for partial redundancy elimination based on SSA form. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*. 273–286.

CHOW, F., CHAN, S., LIU, S., LO, R., AND STREICH, M. 1996. Effective representation of aliases and indirect memory operations in SSA form. In *Proceedings of the Sixth International Conference on Compiler Construction.* 253–267.

CHOW, F., HIMELSTEIN, M., KILLIAN, E., AND WEBER, L. 1986. Engineering a RISC compiler. In *Proceedings of IEEE COMPCON.* 132–137.

CLICK, C. 1995. Global code motion global value numbering. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation.* 246–257.

CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Programming Languages and Systems 13,* 4 (Oct.), 451–490.

DHAMDHERE, D. 1988. A fast algorithm for code movement optimisation. *SIGPLAN Notices 23,* 1, 172–180.

DHAMDHERE, D. 1989. A new algorithm for composite hoisting and strength reduction optimization (+ corrigendum). *Journal of Computer Mathematics 27,* 1–14 (+ 31–32).

DHAMDHERE, D., ROSEN, B., AND ZADECK, K. 1992. How to analyze large programs efficiently and informatively. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.* 212–223.

DHANESHWAR, V. M. AND DHAMDHERE, D. M. 1995. Strength reduction of large expressions. *Journal of Programming Languages 3,* 95–120.

DRECHSLER, K. AND STADEL, M. 1988. A solution to a problem with morel and renvoise's "global optimization by suppression of partial redundancies". *ACM Trans. on Programming Languages and Systems 10,* 4 (Oct.), 635–640.

DRECHSLER, K. AND STADEL, M. 1993. A variation of Knoop, Rüthing and Steffen's lazy code motion. *SIGPLAN Notices 28,* 5 (May), 29–38.

GERLEK, M., STOLTZ, E., AND WOLFE, M. 1995. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form. *ACM Trans. on Programming Languages and Systems 17,* 1 (Jan.), 85–122.

GERLEK, M., WOLFE, M., AND STOLTZ, E. 1994. A reference chain approach for live variables. Tech. Rep. CSE 94-029, Oregon Graduate Institute. Apr.

JOHNSON, R., PEARSON, D., AND PINGALI, K. 1994. The program structure tree: Computing control regions in linear time. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation.* 171–185.

JOSHI, S. M. AND DHAMDHERE, D. M. 1982. A composite hoisting-strength reduction transformation for global program optimization. *International Journal of Computer Mathematics Parts I & II 11,* 1 and 2, 21–41 and 111–126.

KENNEDY, K. 1972. Safety of code motion. *International Journal of Computer Mathematics 3,* 2 and 3, 117–130.

KENNEDY, R., CHOW, F., DAHL, P., LIU, S., LO, R., AND STREICH, M. 1998. Strength reduction via SSAPRE. In *Proceedings of the Seventh International Conference on Compiler Construction.*

KNOOP, J., RÜTHING, O., AND STEFFEN, B. 1992. Lazy code motion. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.* 224–234.

KNOOP, J., RÜTHING, O., AND STEFFEN, B. 1993. Lazy strength reduction. *Journal of Programming Languages 1,* 1 (Mar.), 71–91.

KNOOP, J., RÜTHING, O., AND STEFFEN, B. 1994. Optimal code motion: Theory and practice. *ACM Trans. on Programming Languages and Systems 16,* 4 (Oct.), 1117–1155.

LIU, S., LO, R., AND CHOW, F. 1996. Loop induction variable canonicalization in parallelizing compilers. In *Proceedings of the Fourth International Conference on Parallel Architectures and Compilation Techniques.* 228–237.

LO, R., CHOW, F., KENNEDY, R., LIU, S., AND TU, P. 1998. Register promotion by sparse partial redundancy elimination of loads and stores. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation.*

MOREL, E. AND RENVOISE, C. 1979. Global optimization by suppression of partial redundancies. *Comm. ACM 22,* 2 (Feb.), 96–103.

MUCHNICK, S. 1997. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, San Francisco.

Rosen, B. K., Wegman, M. N., and Zadeck, F. K. 1988. Global value numbers and redundant computations. In *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages*. 12–27.

Schwarz, B., Kirchgässner, W., and Landwehr, R. 1988. An optimizer for Ada – Design, experiences and results. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*. 175–184.

Simpson, L. T. 1996. Value-driven redundancy elimination. Tech. Rep. (PhD Thesis), Dept. of Computer Science, Rice University. Apr.

Sreedhar, V. and Gao, G. 1995. A linear time algorithm for placing ϕ-nodes. In *Conference Record of the Twenty Second ACM Symposium on Principles of Programming Languages*. 62–73.

Wegman, M. and Zadeck, K. 1991. Constant propagation with conditional branches. *ACM Trans. on Programming Languages and Systems 13,* 2 (Apr.), 181–210.

Wolfe, M. 1996. *High Performance Compilers For Parallel Computing*. Addison Wesley.

Received April 1998; accepted September 1998