A New Algorithm for Partial Redundancy Elimination based on SSA Form

Fred Chow Sun Chan Robert Kennedy Shin-Ming Liu Raymond Lo Peng Tu

fchow@sgi.com
Silicon Graphics Computer Systems
2011 N. Shoreline Blvd.
Mountain View, CA 94043

Abstract

A new algorithm, SSAPRE, for performing partial redundancy elimination based entirely on SSA form is presented. It achieves optimal code motion similar to lazy code motion [KRS94a, DS93], but is formulated independently and does not involve iterative data flow analysis and bit vectors in its solution. It not only exhibits the characteristics common to other sparse approaches, but also inherits the advantages shared by other SSA-based optimization techniques. SSAPRE also maintains its output in the same SSA form as its input. In describing the algorithm, we state theorems with proofs giving our claims about SSAPRE. We also give additional description about our practical implementation of SSAPRE, and analyze and compare its performance with a bit-vector-based implementation of PRE. We conclude with some discussion of the implications of this work.

1 Introduction

The Static Single Assignment Form (SSA) has become a popular program representation in optimizing compilers, because it provides accurate use-def relationships among the program variables in a concise form [CFR+91, Wol96, CCL+96]. Many efficient global optimization algorithms have been developed based on SSA. Among these optimizations are dead store elimination [CFR+91], constant propagation [WZ91], value numbering [AWZ88, RWZ88, CS95a], induction variable analysis [GSW95, LLC96], live range computation [GWS94] and global code motion [Cli95]. All these uses of SSA have been restricted to solving problems based on program variables, since the concept of use-def does not readily apply to expressions. Noticeably missing among SSA-based optimizations is partial redundancy elimination.

Partial redundancy elimination (PRE) is a powerful optimization algorithm first developed by Morel and Renvoise [MR79]. By targeting partially redundant computations in the program, it automatically removes global common subexpressions and moves invariant computations out of loops. It has since become the most important component in many global optimizers [Cho83, CHKW86, SKL88, BC94, CS95b]. In [KRS92, KRS94a], Knoop *et al.* formulated an alternative placement strategy called lazy code motion that improves on Morel and Renvoise's results by avoiding unnecessary code movements, and by removing the bidirectional nature of the original PRE data flow equations. The result of lazy code motion is optimal: the number of computations cannot be further reduced by safe code motion, and the lifetimes of the temporaries introduced are minimized. In [DS93], Drechsler and Stadel gave a simpler version of the lazy code motion algorithm that inserts computations on edges rather than in nodes.

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. Because it does not replicate information over the entire program, a sparse representation conserves memory space. Information can be propagated through the sparse representation in a smaller number of steps, speeding up most algorithms. To get the full benefit of sparseness, one must typically give up operating on all elements in the program in parallel, as in traditional bit-vector-based data flow analysis. But operating on each element separately allows optimization decisions to be customized for each object.

There is another advantage of using SSA to perform global optimization. Traditional optimization techniques often implement two separate versions of the same optimization: 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. SSA-based optimization algorithms do not need to distinguish between global and local optimizations. The same algorithm can handle both global and local versions of an optimization simultaneously. The amount of effort required to implement each optimization can be correspondingly reduced.

As was hinted at by Dhamdhere *et al.* in the conclusion of [DRZ92], developing a PRE algorithm based on SSA is difficult because an expression E can be redundant as the result of many different computations at different places of the same expression E', E'', ... whose operands have different SSA versions from the operands of E. This is illustrated in Fig. 1(a). In such a situation, the use-def chain of SSA does little to help in recognizing that E is partially redundant. It also does not help in effecting the movement of computations. Lacking an SSA-based PRE algorithm, optimizers that use SSA have to switch to bit-vector algorithms in performing PRE. To apply subsequent SSA-based opti-

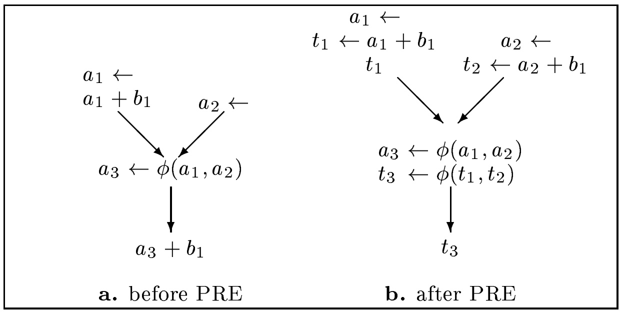

Figure 1: PRE in SSA form

mizations, it is necessary to convert the results of PRE back into SSA form, and such incremental updates based on arbitrary modifications to the program are expensive [CSS96].

We have developed an algorithm that performs PRE directly on an SSA representation of the program (SSAPRE). Our algorithm is sparse because it does not require collecting traditional local data flow attributes over the program and it does not require any form of iterative data flow analysis to arrive at its solution. Our algorithm works by constructing the SSA form of the hypothetical temporary h that could be used to store the result of each computation in the program. In the resulting SSA form of h, a def corresponds to a computation whose result may need to be saved, and a use corresponds to a redundant computation that may be replaced by a load of h. Based on this SSA form of h, we can then apply the analyses corresponding to PRE. The analyses allow us to identify additional defs of h, with accompanying computations, that need to be inserted to achieve optimal code motion. The final output is generated according to the updated SSA graph of h: temporaries are introduced into the program to save and reuse the values of computations. Since the algorithm works by modeling the SSA forms of the hypothetical temporaries, the real temporaries introduced are maintained with SSA properties, as in Fig. 1(b).

The rest of this paper is organized as follows. Section 2 surveys related work aimed at improving the efficiency of data flow analysis and PRE. Section 3 briefly introduces SSA form and gives an overview of the SSAPRE approach. Section 4 describes the SSAPRE algorithm in detail, while stating related lemmas with proofs. Section 5 discusses the theoretical foundations of the SSAPRE algorithm, and verifies its correctness and optimality. Section 6 discusses some practical issues related to an efficient implementation of SSAPRE. Section 7 compares and contrasts the steps in SSAPRE with bit-vector-based PRE, and analyzes the complexity of the SSAPRE algorithm. Section 8 provides measurements that compare the time spent in performing PRE between a bit-vector-based implementation and an implementation of SSAPRE. Section 9 concludes by discussing the implications of this work, and points out some promising areas where similar techniques can be applied using SSAPRE as a model.

2 Related Work

In recent years, we have seen development of different techniques aimed at improving the solution of data flow problems that are related to SSA or PRE.

In [CCF91], by generalizing SSA form, Choi *et al.* derived Sparse Evaluation Graphs as reduced forms of the original flow graph for monotone data flow problems related to variables. The technique must construct a separate sparse graph per variable for each data flow problem, before solving the data flow problem for the variable based on the sparse graph. Thus, it cannot practically be applied to PRE, which requires the solution of several different data flow problems.

In [DRZ92], Dhamdhere *et al.* observed that in solving for a monotone data flow problem, it suffices to examine only the places in the problem where the answer might be different from the trivial default answer $\perp$. There are only three possible transfer functions for a node: raise to $\top$, lower to $\perp$, or identity (propagate unchanged). They proposed *slotwise* analysis. For nodes with the identity transfer function, those that are reached by any node whose answer is $\perp$ will have $\perp$ as their answer. By performing the propagation slotwise, the method can arrive at the solution for each variable in one pass over the control flow graph. Slotwise analysis is not sparse, because it still performs the propagation with respect to the control flow graph of the program. The approach can be used in place of the iterative solution of any monotone data flow problem as formulated. It can be used to speed up the data flow analyses in PRE.

In [Joh94], Johnson proposed the use of Dependence Flow Graphs (DFG) as a sparse approach to speed up data flow analysis. The DFG of a variable can be viewed as its SSA graph with additional "merge" operators imposed to identify single-entry single-exit (SESE) regions for the variable. By identifying SESE regions with the identity transfer function, the technique can short-circuit propagation through them. Johnson showed how to apply his techniques to the data flow systems in Drechsler and Stadel's variation of Knoop *et al.*'s lazy code motion.

Researchers at Rice University have done work aimed at improving the effectiveness of PRE [BC94, CS95b]. The work involves the application of some SSA-based transformation techniques to prepare the program for optimization by PRE. Their techniques enhance the results of PRE. Their implementation of PRE was based on Drechsler and Stadel's variation of Knoop *et al.*'s lazy code motion, and was unrelated to SSA.

All prior work related to PRE has modeled the problem as systems of data flow equations. Regardless of how efficiently the systems of data flow equations can be solved, a substantial amount of time needs to be spent in scanning the contents of each basic block in the program to initialize the local data flow attributes that serve as input to the data flow equations. Experience has shown that this often takes more time than the solution of the data flow equations, so a fundamentally new approach to PRE that does not require the dense initialization of data flow information is highly desirable. SSAPRE satisfies this property as it exploits sparseness.

3 Overview of Approach

The input to SSAPRE is an SSA representation of the program. In SSA, each definition of a variable is given a unique version, and different versions of the same variable can be regarded as different program variables. Each use of a variable version can only refer to a single reaching definition. By virtue of the versioning, use-def information is built into the representation. Where several definitions of a variable, $a_1, a_2, \ldots, a_m$, reach a confluence point in the control flow graph of the program, a ϕ function assignment statement, $a_n \leftarrow \phi(a_1, a_2, \ldots, a_m)$, is inserted to merge them into the definition of a new variable version a_n. Thus the semantics of single reaching definitions is maintained. This introduction of a new variable version as the result of ϕ factors the set of use-def edges over confluence nodes, reducing the number of use-def edges required to represent the program. In SSA, the use-def chain for each variable can be provided by making each version point to its single definition. One important property of SSA form is that each definition must dominate all its uses in the control flow graph of the program if the uses at ϕ operands are regarded as occurring at the predecessor nodes of their corresponding edges.

We assume all expressions are represented as trees with leaves that are either constants or SSA-renamed variables. SSAPRE can be applied to program expressions independently, regardless of subexpression relationships. In Section 6, 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 indirect variables have to be in SSA form in order for SSAPRE to handle them. Using the HSSA form presented in [CCL$^+$96] allows SSAPRE to uniformly handle indirect loads together with other expressions in the program.

SSAPRE consists of six separate steps: (1) *Φ-Insertion*, (2) *Rename*, (3) *DownSafety*, (4) *WillBeAvail*, (5) *Finalize* and (6) *CodeMotion*. SSAPRE works by conducting a round of SSA construction on the lexically identical expressions in the program whose variables are already in SSA form.[1] Since the term SSA cannot be meaningfully applied to expressions, we define it to refer to the hypothetical temporary h that could be used to store the result of the expression. In the rest of this paper, we use Φ to refer to a ϕ in the SSA form of the hypothetical temporary to contrast it with a ϕ for a variable in the original program.

Φ-Insertion and *Rename* are the initial SSA construction steps for expressions. This round of SSA construction can use an approach similar to that described in [CFR$^+$91], working on all expressions in the program simultaneously. Alternatively, an implementation may choose to work on each lexically identical expression in sequence. We describe such a sparse implementation in Section 6.

Assuming we are working on the expression $a + b$, whose hypothetical temporary is h. After the *Rename* step, occurrences of $a + b$ corresponding to the same version of h must compute the same value. At this stage, the points of defs and uses of h have not yet been identified. Many Φ's inserted for h are also unnecessary. Later steps in SSAPRE will fix them up. Some Φ operands can be determined to be undefined ($\perp$) after *Rename* because there is no available computation of $a + b$. These $\perp$-valued Φ operands will play a key role in the later steps of SSAPRE, because insertions are performed only because of them. We call the SSA graph[2] for h after *Rename* the *dense SSA graph* because it contains more Φ's than in the minimal SSA form (as defined in [CFR$^+$91]).

[1] Expressions are *lexically identical* if they apply exactly the same operator to exactly the same operands; the SSA versions of the variables are ignored in matching expressions. For example, $a_1 + b_1$ and $a_2 + b_2$ are lexically identical expressions.

[2] Our SSA graph is similar to that described in [GSW95], which is formed from the use-def edges of nodes assigned the same SSA version.

The sparse computation of global data flow attributes for $a + b$ can be performed on the dense SSA graph for h. Two separate phases are involved. The first phase, *DownSafety*, performs backward propagation to determine the Φ's whose results are not fully anticipated with respect to $a + b$. The second phase is *WillBeAvail*, which performs forward propagation to determine the Φ's where the computation of $a + b$ will be available assuming PRE insertions have been performed at the appropriate incoming edges of the Φ's.

Using the results of *WillBeAvail*, we are ready to finalize the effects of PRE. The *Finalize* step inserts computation of $a + b$ at the incoming edges of Φ to ensure that the computation is available at the merge point. For each occurrence of $a + b$ in the program, it determines if it is a def or use of h. It also links the uses of h to their defs to form its *precise SSA graph*. Extraneous Φ's (see [CFR$^+$91], p.359) are removed so that h is in minimal SSA form.

The last step is to update the program to effect the code motion for $a + b$ as determined by SSAPRE. The *CodeMotion* step introduces the real temporary t to eliminate the redundant computations of $a + b$. It walks over the precise SSA graph of h and generates saves of the computation $a + b$ into t, giving each t its unique SSA version. Redundant computations of $a + b$ are replaced by t. The Φ's for h are translated into ϕ's for t in the native program representation.

4 SSAPRE Algorithm

In this section, we describe the complete SSAPRE algorithm. As in [KRS92] and [DS93], we assume all *critical edges* in the control flow graph have been removed by inserting empty basic blocks at such edges. This allows us to model insertions as edge placements, even though we only insert at the ends of the predecessor blocks.

We assume prior computation of the dominator tree (DT) and dominance frontiers (DF's) with respect to the control flow graph of the program. These data must have already been computed and used when the program was first put into SSA form [CFR$^+$91]. Again, we base our discussion on the expression $a + b$ whose hypothetical temporary is h. We use the example program shown in Fig. 2 to illustrate the various steps. Based on the algorithms we describe, we also state and prove various lemmas, which we use in establishing the theorems about SSAPRE in Section 5.

4.1 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 frontiers (DF$^+$), because the occurrence may correspond to a def of h. In Fig. 3, a Φ is inserted at block 3 due to $a + b$ in block 1.

The second situation that causes insertion of Φ's is when there is a ϕ for a variable contained in the expression, because that indicates an alteration of the expression reaches the merge point. We only need to insert a Φ at a merge point when it reaches a later occurrence of the expression, because otherwise the Φ will not contribute to any optimization in PRE. In Fig. 3, the Φ for h at block 8 is caused by the ϕ for a in the same block. We do not need to insert any

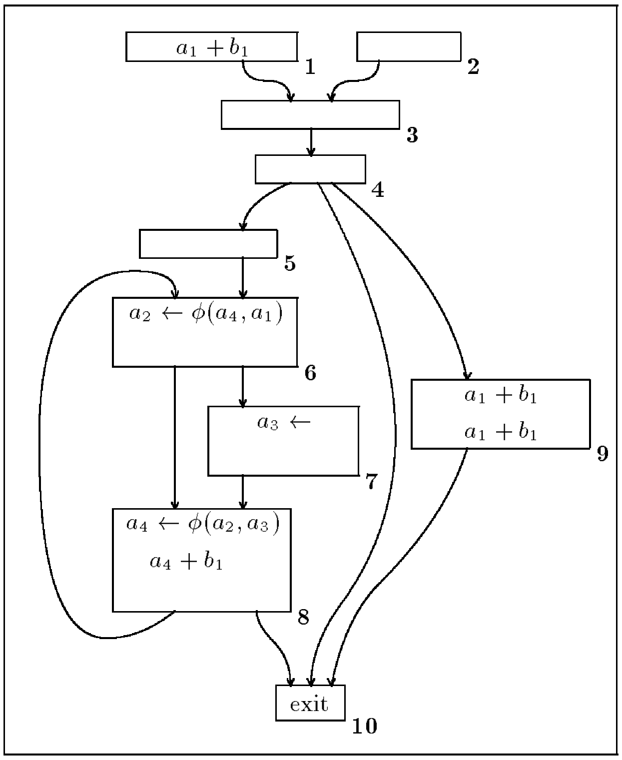

Figure 2: Example Program (in SSA form)

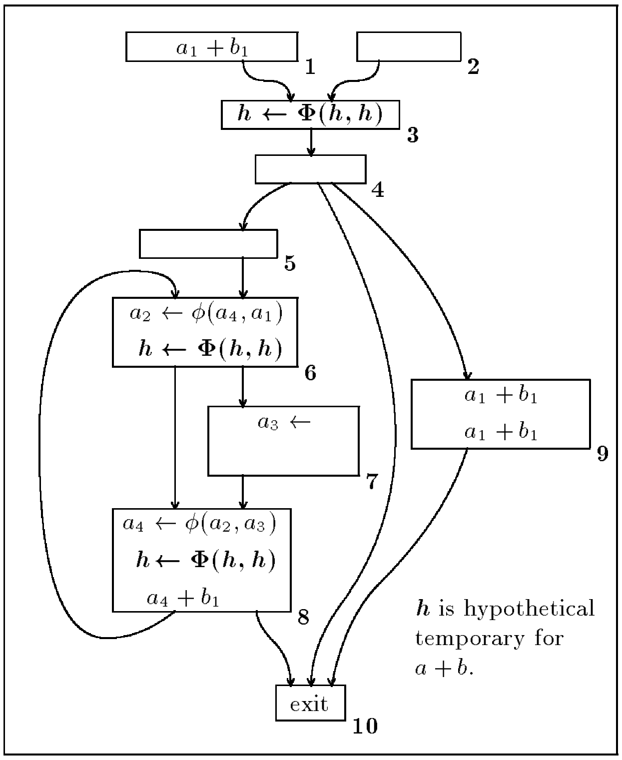

Figure 3: Program after Φ-Insertion

Φ at block 10 even though it is a merge point, because there is no later occurrence of $a + b$ after block 10.

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 due to DF^+ 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 contribute to 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 recursively check 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 Fig. 4. By using this demand-driven technique, we take advantage of the SSA representation in the input program.

Other algorithms for SSA ϕ placement with linear time complexity can also be used to place Φ's [JPP94, SG95]. We adapt the algorithm from [CFR+91] because it is easier to understand and implement.

LEMMA 1 *(Sufficiency of Φ insertion) If B is a basic block where no expression Φ is inserted and the expression is partially anticipated at the entry to B, exactly one evaluation of the expression (counting $\perp$ as an evaluation) can reach the entry to B.*

Proof: Suppose at least 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 computation of the expression, 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 B_0, it must have placed a Φ there, once again contradicting the proposition that ψ_1 reaches B. $\square$

4.2 The *Rename* Step

The *Rename* step assigns SSA versions to h in its SSA form. The version numbering we produce for h differs from the eventual SSA form for the temporary t, but has the following two important properties. First, occurrences that have identical h-versions have identical values. Second, any control flow path that includes two different h-versions must cross an assignment to an operand of the expression or a Φ for h.

We apply the SSA Renaming algorithm as given in [CFR+91], 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 we back up the blocks that define them. Maintaining the variable and expression stacks together allows us to decide efficiently whether two occurrences of an expression should be given the same h-version.

```
procedure Φ-Insertion
  for each expression E_i do {
    DF_phis[i] ← empty-set
    for each variable j in E_i do
      Var_phis[i][j] ← {}
  }
  for each occurrence X of E_i in program do {
    DF_phis[i] ← DF_phis[i] ∪ DF⁺(X)
    for each variable occurrence V in X do
      if (V is defined by φ) {
        j ← index of V in X
        Set_var_phis(Phi(V), i, j)
      }
  }
  for each expression E_i do {
    for each variable j in E_i do
      DF_phis[i] ← DF_phis[i] ∪ Var_phis[i][j]
    insert Φ's for E_i according to DF_phis[i]
  }
end Φ-Insertion

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

Figure 4: Algorithm for Φ-*Insertion*

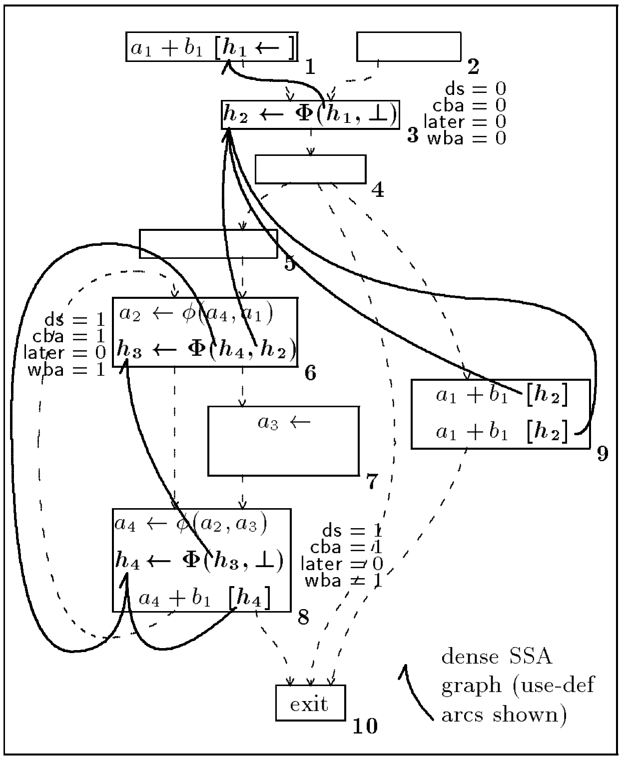

Figure 5: The dense SSA graph for $a + b$

There are three kinds of occurrences of expressions in the program: (1) the expressions 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 exits of the predecessor nodes of 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 version. 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 version as the top of E_i's stack. 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 version; (b) if q is a Φ operand, we assign the special version $\perp$ 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. Fig. 5 shows the dense SSA graph that forms after h in our example has been renamed. This expression renaming technique also takes advantage of the SSA representation of the program variables.

The remaining steps of the SSAPRE algorithm rely on the fact that Φ's are placed only where E_i is partially anticipated, (*i.e.*, there is no dead Φ in the SSA graph of h). Dead Φ's can efficiently be identified by applying the standard SSA-based dead store elimination algorithm [CFR+91] on the SSA graph formed after renaming. From here on, we assume that only live Φ's are represented in the SSA form of h.

LEMMA 2 *(Correctness of version renaming) If two occurrences are assigned the same version 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 the same version to two occurrences of an expression E_i 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 [CFR+91] to complete the proof. □

LEMMA 3 *(Versions capture all the redundancy) If two occurrences ψ_x, ψ_y are assigned versions x, y by* Rename, *exactly one of the following holds:*

- *no control flow path can reach from ψ_x to ψ_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*
- *there is a path (possibly empty, in which case $x = y$) in the SSA graph of use-def arcs from y to x (implying that any redundancy between ψ_x and ψ_y is exposed to the algorithm).*

Proof: Suppose there is a control flow path $\mathcal{P}$ from ψ_x to ψ_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 ψ_y. Now we apply the induction hypothesis to that part of $\mathcal{P}$ up to the corresponding operand of that Φ. □

4.3 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 [KRS94a]. In the dense SSA graph 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 only necessary at Φ's, so down-safety only needs to be computed for them. Using the SSA graph, down-safety can be sparsely computed by backward propagation along the use-def 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 result version is not used; or (b) there is a path to exit along which the only use of the Φ result version is as an operand of a Φ that is not down-safe. Case (a) represents the initialization for our backward propagation of down-safety; all other Φ's are initially marked *down_safe*. *DownSafety* propagation is based on case (b). Since a real occurrence of the expression blocks the case (b) propagation, the algorithm marks each Φ operand with a flag *has_real_use* when the path to the Φ operand crosses a real occurrence of the same version of the expression.

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 SSA graph. 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 version to a real occurrence of expression E_i or encounters a program 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 version it defines is not used along the path to the current occurrence (or exit). When *Rename* assigns a version to a Φ operand, it sets that operand's *has_real_use* flag if and only if a real occurrence for the same version appears at the top of the rename stack.

Fig. 6 gives the *DownSafety* propagation algorithm.

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 version 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 version 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. Minimality of the input HSSA program implies, then, that any path from the Φ to the current occurrence and continuing to a program exit must encounter an assignment to ν before encountering an evaluation of the expression. Therefore the expression is not fully anticipated at the Φ.

Next we make the observation that any Φ whose *down_safe* flag gets cleared during the *DownSafety* step is not down-safe, since there is a path in the SSA use-def graph

procedure *DownSafety*
  for each expr-Φ F in program **do**
    if (**not** *down_safe(F)*)
      for each operand *opnd of F* **do**
        Reset_downsafe(opnd)
end *DownSafety*

procedure *Reset_downsafe(X)*
  if (*has_real_use(X)* **or** X not defined by Φ)
    return
  $F \leftarrow \Phi$ that defines X
  if (**not** *down_safe(F)*)
    return
  down_safe(F) $\leftarrow$ false
  for each operand *opnd* of F **do**
    Reset_downsafe(opnd)
end *Reset_downsafe*

Figure 6: Algorithm for *DownSafety*

from an unused version to that Φ where no arc 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*. $\Box$

4.4 The *WillBeAvail* Step

The *WillBeAvail* step has the task of predicting whether the expression will be available at each Φ result 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.

The *WillBeAvail* step consists of two forward propagation passes performed sequentially, in which we conduct simple reachability search in the SSA graph 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 and the false value is propagated from such nodes to others that are not down-safe and that are reachable along def-use arcs in the SSA graph, excluding arcs at which *has_real_use* is true. Φ operands defined by Φ's that are not *can_be_avail* are set to $\perp$ along the way. 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.[3]

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

[3] 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 [KRS92] and [DS93] because their bit-vector schemes allow earliest insertion at non-merge blocks.

points. The second pass is analogous to the computation of the predicate LATERIN in [DS93]. It works by propagating the *later* predicate, which it initializes to true wherever *can_be_avail* is true. It then begins with the real occurrences of the expression in the program, and propagates the false value of *later* forward to those points beyond which insertions cannot be postponed (moved downward) without introducing unnecessary new redundancy.

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

$$will\_be\_avail = can\_be\_avail \land \neg later.$$

Fig. 5 shows the values of *down_safe* (ds), *can_be_avail* (cba), *later* and *will_be_avail* (wba) for the program example at each Φ for h. 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*.

Fig. 7 gives the *WillBeAvail* propagation algorithms.

As in [KRS92], we use the term *placement* to refer 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 insertions makes the expression available immediately after the Φ.*

Proof: Let 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 def-use arcs in the SSA graph.

Now let 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* gives a path bearing witness to the fact that no safe set of insertions can make the expression available at F. □

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 that Φ's result is not available immediately after the Φ.*

Proof: The set of Φ's not satisfying *later* after *WillBeAvail* is exactly the set of *can_be_avail* Φ's reachable along def-use arcs in the SSA graph from *has_real_use* operands of *can_be_avail* Φ's. Let $\mathcal{P}$ be a path in the def-use SSA graph from such a Φ operand to a given expr-Φ F with $later(F) =$ false. 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 arcs there is a *has_real_use* operand of F. Such an operand must

```
procedure Compute_can_be_avail
  for each expr-Φ F in 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_can_be_avail(G)
  can_be_avail(G) ← false
  for each expr-Φ F with operand opnd defined by G do
    if (not has_real_use(opnd)) {
      set that Φ operand to ⊥
      if (not down_safe(F) and can_be_avail(F))
        Reset_can_be_avail(F)
    }
end Reset_can_be_avail

procedure Compute_later
  for each expr-Φ F in program do
    later(F) ← can_be_avail(F)
  for each expr-Φ F in program do
    if (later(F)) and
        ∃ an operand opnd of F such that
          (opnd ≠ ⊥ and has_real_use(opnd)))
      Reset_later(F)
end Compute_later

procedure Reset_later(G)
  later(G) ← false
  for each expr-Φ F with operand opnd defined by G do
    if (later(F))
      Reset_later(F)
end Reset_later

procedure WillBeAvail
  Compute_can_be_avail
  Compute_later
end WillBeAvail
```

Figure 7: Algorithm for *WillBeAvail*

be fully available in the optimized program, so any insertion below F would be redundant with that operand, contradicting computational optimality. Since F is down-safe, that operand is already redundant with real occurrence(s) in the unoptimized program 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 arc, we apply the induction hypothesis to the Φ defining the operand of F corresponding to the final arc 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). □

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 set of insertions chosen by SSAPRE together 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 expr-Φ F, if there is a path $\mathcal{P}(F)$ of length k in the SSA def-use graph beginning with $\perp$, passing only through Φ's that are not *will_be_avail* along arcs 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 no insertion is performed for any operand of F, since it 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 arc 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 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 expr-Φ 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 def-use arcs 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. $\square$

4.5 The *Finalize* Step

The *Finalize* step plays the role of transforming the SSA graph for the hypothetical temporary h to the valid SSA form that reflects insertions and in which no Φ operand is $\perp$. The *Finalize* step performs the following tasks:

- It decides for each real occurrence of the expression whether it should be computed on the spot or reloaded from the temporary. For each one that is computed, it also decides whether the result should be saved to the temporary. It sets two flags, *reload* and *save*, to represent these two pieces of information.
- 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.
- Expression 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 links from *will_be_avail* Φ's that reference them are fixed up to refer to other (real or inserted) occurrences.
- Extraneous Φ's are removed.

Finalize creates a table $Avail\_def_i$ (for *available definitions*) for each expression E_i to perform the first three of the above tasks. The indices into this table are the SSA versions for E_i's hypothetical temporary h. $Avail\_def_i[x]$ will point to the defining occurrence of E_i for h_x, which must be either: (a) a real occurrence, or (b) a Φ for which *will_be_avail* is true. *Finalize* performs a preorder traversal of the dominator tree of the program control flow graph. In the course of this traversal it will visit each defining 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 h_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 [CFR+91], *Finalize* does not require any renaming stack because SSA versions have already been assigned.

In the course of its traversal, *Finalize* will process occurrences as follows:

1. Φ — If its *will_be_avail* is false, nothing needs to be done. (An example of this is the Φ in block 3 of our running example. See Fig. 5.) Otherwise, we must be visiting h_x for the first time. Set $Avail\_def_i[x]$ to this Φ.
2. Real occurrence of E_i — If $Avail\_def_i[x]$ is $\perp$, we are visiting h_x the first time. If $Avail\_def_i[x]$ is set, but that occurrence does not dominate the current occurrence, the current occurrence is also a definition of h_x. (An example of this latter case is the first h_2 in block 9 of our example.) In both of these cases, we update $Avail\_def_i[x]$ to the current occurrence. Otherwise, the current occurrence is a use of h_x, and we set the *save* flag in the occurrence pointed to by $Avail\_def_i[x]$ and the *reload* flag of the current occurrence.
3. Operand of Φ in a successor block[4] — If *will_be_avail* of the Φ is false, nothing needs to be done. Otherwise if the operand satisfies *insert*, (*e.g.*, operand h_2 in the Φ at block 6 of our example), insert a computation of E_i at the exit of the current block. If *will_be_avail* holds but the operand does not satisfy *insert*, set the *save* flag in the occurrence pointed to by $Avail\_def_i[x]$ (which cannot be $\perp$), and update that Φ operand to refer to $Avail\_def_i[x]$ (e.g. operand h_3 in the Φ at block 8 of our example).

The full algorithm to perform the above tasks is given in Fig. 8.

The removal of extraneous Φ's, or SSA minimization, for h is not a necessary task as far as PRE is concerned. However, the extraneous Φ's take up storage in the program representation, and may affect the efficiency of other SSA-based optimizations to be applied after PRE. Removing extraneous Φ's also requires changing their uses to refer to their replacing versions. SSA minimization can be implemented as a variant of the ϕ insertion step in SSA construction [CFR+91, JPP94, SG95]. 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 assignments to h in the program (*i.e.*, real occurrences with the *save* bit set plus the inserted computations). We then pass over all the extraneous Φ's to determine a *replacing version* for each one. Whenever an extraneous Φ defines version h_x and has an operand using h_y that is not defined by an extraneous Φ, y is the replacing version for x. From such a Φ we propagate the replacing version through all its uses: once the replacing version for a Φ is known, the replacing version for every use of that Φ becomes known (the replacing version of each use is the same as the replacing version of the Φ) and we propagate recursively to all uses of that Φ. It straightforward

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

```
procedure Finalize_visit(block)
   for each occurrence X of E_i in block do {
      save(X) ← false
      reload(X) ← false
      x ← version(X)
      if (X is Φ) {
         if (will_be_avail(X))
            Avail_def[i][x] ← X
      }
      else if (Avail_def[i][x] is ⊥ or
               Avail_def[i][x] does not dominate X)
         Avail_def[i][x] ← X
      else if (Avail_def[i][x] is real) {
         save(Avail_def[i][x]) ← true
         reload(X) ← true
      }
   }
   for each S in Succ(block) do {
      j ← WhichPred(S, block)
      for each expr-Φ F in S do
         if (will_be_avail(F)) {
            i ← WhichExpr(F)
            if (j^th operand of F satisfies insert) {
               insert E_i at the exit of block
               set j^th operand of F to inserted occurrence
            }
            else {
               x ← version(j^th operand of F)
               if (Avail_def[i][x] is real) {
                  save(Avail_def[i][x]) ← true
                  set j^th operand of F to Avail_def[i][x]
               }
            }
         }
   }
   for each K in Children(DT, block) do
      Finalize_visit(K)
end Finalize_visit

procedure Finalize
   for each version x of E_i in program do
      Avail_def[i][x] ← ⊥
   Finalize_visit(Root(DT))
end Finalize
```

Figure 8: Algorithm for *Finalize*

to see that this method replaces all references to extraneous Φ's by references to non-extraneous occurrences.

Fig. 9 shows our example program at the end of the *Finalize* step.

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 versions while traversing the SSA graph in dominator-tree preorder. In particular, *Finalize* ensures directly that each reload is dominated by its available definition. Because the live ranges of different versions of h do not overlap, each reloaded occurrence must refer to its available definition. □

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

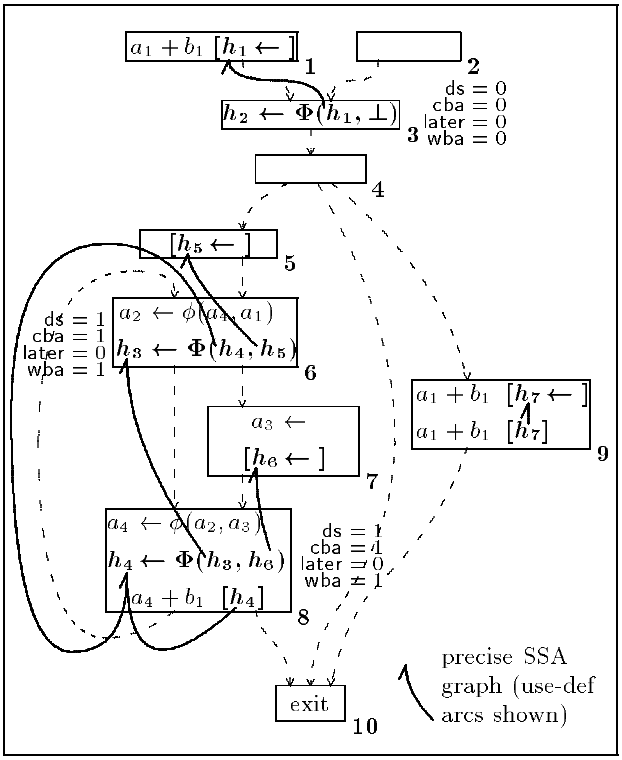

Figure 9: Program after *Finalize*

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 of the same version. 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. □

4.6 The *CodeMotion* Step

Once the hypothetical temporary h has been put into valid SSA form, the only remaining task is to update the SSA program representation to reflect the results of PRE. This involves introducing the real temporary t for the purpose of eliminating redundant computations. This task is straightforward due to the fact that h is already in valid SSA form. The SSA form of t is a subgraph of the SSA form of h, since defs of h (including Φ's) with no use are omitted.

The *CodeMotion* step walks over the SSA graph of h. At a real occurrence, if *save* is true, it generates a save of the result of the computation into a new version of t. If *reload* is true, it replaces the computation by a use of t. At an inserted occurrence, it saves the value of the inserted computation into a new version of t. At a Φ of h, it generates a corresponding ϕ for t. Fig. 10 shows our example program at the end of the *CodeMotion* step.

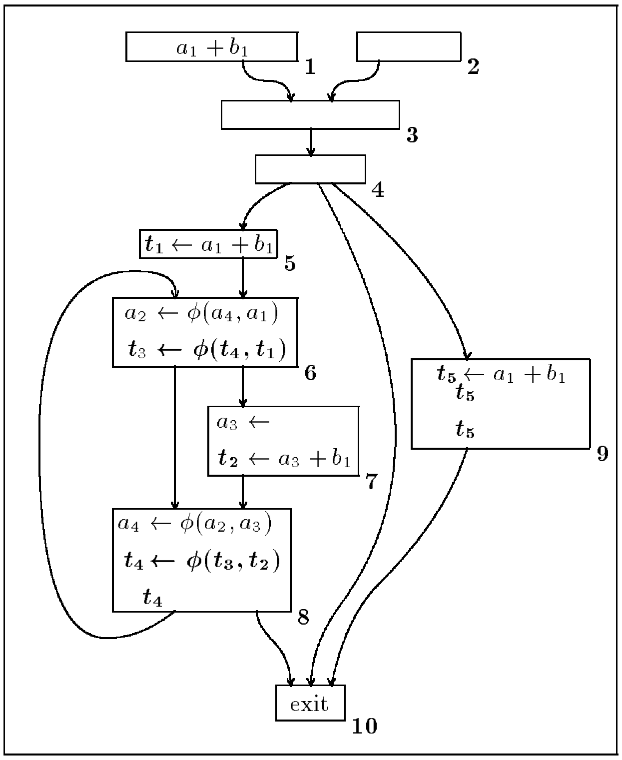

Figure 10: Program after *CodeMotion*

5 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 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 reloads are generated only when the reloaded occurrence is dominated by a *will_be_avail* Φ of the same version (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 version 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 that occurrence dominates a use of the same version by another real occurrence or by a Φ operand. In the former case the result is immediate, and in the latter case we need only appeal to the fact that the expression is partially anticipated at every Φ remaining after the *Rename* step. □

THEOREM 4 *SSAPRE chooses a* computationally optimal *placement;* i.e., *no safe placement can result in fewer computations 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 version (by Lemma 9) nor is it defined by a *will_be_avail* Φ (by Lemma 7). Because ψ_1 and ψ_2 do not have the same version and there is no assignment to any expression operand along $\mathcal{P}$, the definition of ψ_2's version 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 C denotes any computationally optimal placement, 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 of the dominance-frontier ϕ-insertion algorithm. 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. □

6 Practical Implementation

Since SSAPRE is a sparse algorithm, an implementation can reduce the maximum storage needed to optimize all the expressions in the program by finishing the work on each expression before moving on to the next one. Under this scheme, the different lexically identical expressions that need to be worked on by SSAPRE are maintained as a worklist. If the expressions in the program are represented in tree form, we can also exploit the nesting relationship in expression trees to reduce the overhead in the optimization of large expressions. There is also a more efficient algorithm for performing the *Rename* step of SSAPRE. In this section, we give a brief description of these implementation techniques.

6.1 Worklist-driven PRE

Under worklist-driven PRE, we add an initial pass, *Collect-Occurrences*, that scans the entire program and creates a worklist for all the expressions in the program that need to be worked on by SSAPRE. For each element of the worklist, we represent its occurrences in the program as a set of *occurrence nodes*. Each occurrence node provides enough information to pinpoint the location of the 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 expression based only on its occurrence nodes. The intermediate storage needed to work on each expression can be reclaimed when working on the next one.

Collect-Occurrences enters only *first order* expressions into the worklist. First order expressions contain only one operator. For example, in the expression $(a+b)-c$, $a+b$ is the first order expression and is entered into the worklist, but $(a+b)-c$ is not initially entered into the worklist. After SSAPRE has worked on $a+b$, any redundant occurrence of $a+b$ will be replaced by a temporary t. If PRE on $a+b$ changes $(a+b)-c$ to $t-c$, the *CodeMotion* step will enter the new first order expression $t-c$ as a new member of the worklist. Redundant occurrences of $t-c$, and hence redundancies in $(a+b)-c$, will be replaced when $t-c$ is processed. If the expression $(a+b)-c$ does not yield $t-c$ when $a+b$ is being worked on, $a+b$ is not redundant, implying that $(a+b)-c$ has no redundancy and can be skipped by SSAPRE. This approach deals cleanly with the interaction between the optimizations of nested expressions and gains efficiency by ignoring the higher order expressions that exhibit no redundancy.[5] This strategy is hard to implement in bit-vector PRE, which typically works on all expressions in the program simultaneously in order to take advantage of the parallelism inherent in bit-vector operations.

In manipulating the sparse representation of each expression, some steps in the algorithm need to visit the occurrence nodes in an order corresponding to a preorder traversal of the dominator tree of the control flow graph. For this purpose, we maintain the occurrence nodes for a given expression in the order of this preorder traversal of the dominator tree. As we mentioned in Section 4.2, there are three kinds of occurrences. *Collect-Occurrences* only creates the real occurrence nodes. The Φ-*Insertion* step inserts new occurrence nodes that represent Φ's and Φ operands. Under worklist-driven PRE, we need a fourth kind of occurrence nodes to indicate when we reach the program exits in the *Rename* step. These *exit* occurrence nodes can be represented just once and shared by all expressions. Fig. 11 is a flow chart for our SSAPRE implementation.

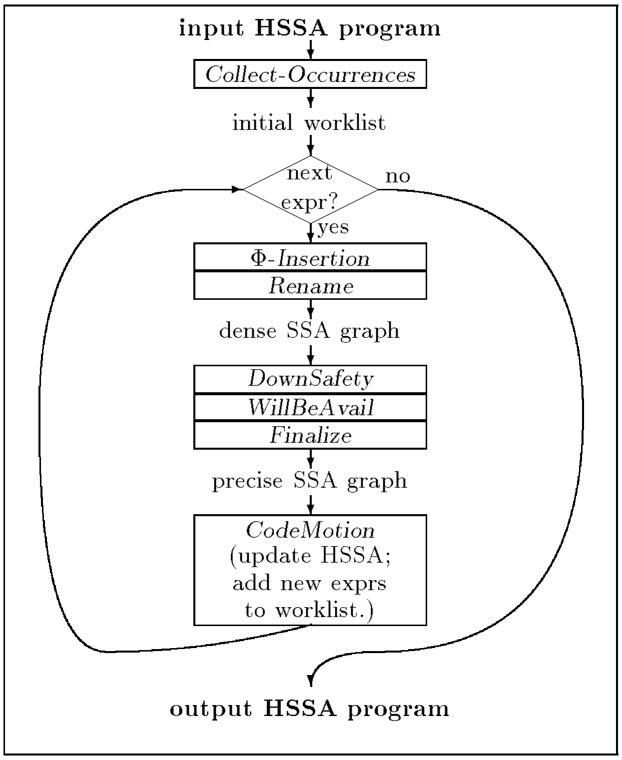

Figure 11: SSAPRE implementation flow chart

6.2 Delayed Renaming

The *Rename* algorithm described in Section 4.2 maintains version stacks for all the variables in the program in addition to the version stacks for the expressions. Apart from taking up additional storage, updating the variable stacks requires keeping track of when the values of the variables change, which may incur significant overhead. The algorithm is not in line with sparseness, because in a sparse algorithm, the time spent in optimizing an expression should not be affected by the number of times its variables are redefined. Also, under the worklist-driven implementation of SSAPRE, we can no longer pass over the entire program in the *Rename* step, because that would imply passing over the entire program once for every expression in the program. The solution of both of these problems is to use a more efficient algorithm for renaming called *delayed renaming*.

Recall the purpose of the variable stacks in the *Rename* step is to enable us to determine when the value of an available expression is no longer current by checking if the versions of all the variables are the same as the current versions. At a real occurrence of the expression, we do not have to rely on the variable stacks, because the current versions of all its variables are represented in the expression. We only need the variable stacks when renaming Φ operands.

To implement delayed renaming, the *Rename* step is replaced by two separate passes. The first pass, *Rename-1*, is the same as *Rename*, except that it does not use any variable stack. At a Φ operand, it optimistically assumes that its version is the version on top of the expression stack. Thus, it can perform all its work based on the occurrence nodes of the expression. *Rename-1* computes an initial version of the SSA graph for h that is *optimistic* and not entirely correct. The correct renaming of Φ operands is *delayed* to the second pass, *Rename-2*, which relies on seeing a later real occurrence of the expression to determine the current versions of the variables. Seeing a later real occurrence implies that at the earlier Φ, the expression is partially anticipated. Thus, the versions of the Φ operands are fixed up only for these Φ's.

Rename-2 works according to a worklist built for it by *Rename-1*, which contains all the real occurrences that are defined by Φ's. From the versions of the variables at the merge block of a Φ, it determines the versions of the variables at each predecessor block based on the presence or absence of ϕ's for the variables at that merge block. If they are different from the versions assumed at the Φ operand in the *Rename-1* pass, *Rename-2* invalidates the Φ operand by

[5] For higher order expressions that have redundancies, this approach also has the secondary effect of converting the expression tree essentially to triplet form.

Rule	def at top of stack	current occurrence X	condition for identical h-version
1	real	real	all corresponding variables have same versions
2	real	Φ operand	
3	Φ	real	defs of all variables in X dominate the Φ
4	Φ	Φ operand	

Table 1: Assigning h-versions in Delayed Renaming

resetting it to $\perp$. Otherwise, the Φ operand renamed by *Rename-1* is correct. If the Φ operand is also defined by Φ, it is added to the worklist so that the process can continue up the SSA graph. For example, *Rename-1* will initially set the second operand of the Φ for h in block 8 of Fig. 5 to h_3. *Rename-2* resets it to $\perp$.

Table 1 gives the rules for deciding when two occurrences should be assigned the same h-version in the absence of the variable stacks. Rules 1 and 3 are applied in *Rename-1*, while rules 2 and 4 are applied in *Rename-2*.

An additional advantage of delayed renaming is that it allows us to determine the Φ's that are not live without performing a separate dead store elimination phase. In delayed renaming, only the operands at Φ's at which the expression is partially anticipated are fixed up. The remaining Φ's correspond to dead Φ's, and they can be marked for deletion.

7 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 SSA graph for the hypothetical temporary 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. The SSA graph of the hypothetical temporary also allows SSAPRE to easily maintain the generated temporary in SSA form.

The complexities of the various steps in SSAPRE can be easily established. Assuming the implementation described in Section 6, 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 SSA graph. The *Φ-Insertion* step is $\Omega(v^2)$ for insertion at domination frontiers, but as we explained in Section 4.1, 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.

8 Measurements

We have implemented SSAPRE in WOPT, the global optimizer in the Silicon Graphics MIPSpro Compilers. The optimizer uses a variant of SSA called HSSA as its internal program representation [CCL+96]. The optimizer had used the bit-vector-based Morel and Renvoise algorithm [Cho83] to perform PRE, while it uses known SSA-based algorithms for its other optimizations. In Release 7.2 of the compiler, we have re-implemented the PRE phase using SSAPRE, incorporating the techniques we described in Section 6. In this section, we compare their performance differences using the SPECint95 and SPECfp95 benchmark suites.

In terms of optimization results, measured by the running time of the benchmarks, 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 start out 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. 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. Table 2 gives our timing results as measured on a 195 MHz R10000 Silicon Graphics Power Challenge. The benchmarks were compiled under the optimization level -O2, which does not invoke procedure inlining.

The measurements in Table 2 show widely different results across the various benchmarks. In the SPECint95 benchmarks, SSAPRE ranges from 65% faster in **perl** to 29% slower 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

SPECint95 Benchmarks	go	m88ksim	gcc	compress	li	ijpeg	perl	vortex
Bit-vector PRE (T1)	116900	4850	886360	100	12950	10340	98840	62950
SSAPRE (T2)	151260	4440	339160	60	5090	11200	34970	53000
Ratio T2/T1	1.293	0.915	0.382	0.600	0.393	1.083	0.353	0.841

SPECfp95 Benchmarks	tomcatv	swim	su2cor	hydro2d	mgrid	applu	turb3d	apsi	fpppp	wave5
Bit-vector PRE (T1)	40	170	500	7080	500	5060	2420	37930	1450	94150
SSAPRE (T2)	60	400	700	8780	1400	9450	5000	93960	1980	85800
Ratio T2/T1	1.500	2.352	1.399	1.240	2.799	1.867	2.066	2.477	1.365	0.911

Table 2: Time (in msec.) spent in Partial Redundancy Elimination in compiling SPECint95 and SPECfp95

analysis 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.

There is still work to be done in tuning the implementation of SSAPRE. Using a characterization of the common sizes and forms of SSA graphs of the hypothetical temporary, we expect to improve the implementation of many parts of the algorithm to speed up SSAPRE's processing. Investigation into SSAPRE's wide compile-time performance differences relative to bit-vector-based PRE may offer insights that lead to more efficient implementation.

9 Conclusion and Further Work

The SSAPRE algorithm presented in this paper performs PRE while taking full advantage of the SSA form in the input program and within its operation. It incorporates the advantages shared by all the other SSA-based optimization techniques: no separate phase to collect local attributes, no data flow analysis involving bit vectors, sparse representation, sparse computation of global attributes, and unified handling of each optimization's global and local forms. In actual implementation, by working on one expression at a time, we can also lower the maximum storage requirement needed to optimize all the expressions in the program, and also exploit the nesting relationship in expression trees to speed up the optimization of large expressions.

SSAPRE enables PRE to be seamlessly integrated into a global optimizer that uses SSA as its internal representation. Because the SSA form is updated as optimization progresses, optimizations can be re-invoked as needed without incurring the cost of repeatedly rebuilding SSA. From an engineering point of view, SSAPRE permits a cohesive software implementation by making SSA and sparseness the theme throughout the optimizer.

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 or operations in the program. This work shows that data flow problems for expressions can be modeled in SSA form by introducing hypothetical temporaries that store the values of expressions. Such an approach opens up new ways to solve many data flow problems by first formulating their solution in terms of the SSA graph of the hypothetical temporary. Candidates for this new approach are code hoisting and the elimination of load and store redundancies [Cho88, KRS94b]. We intend to pursue such work in the near future.

The SSAPRE approach can also incorporate techniques developed in the context of classical PRE, such as the integration of strength reduction into the PRE optimization phase [Cho83, Dha89, KRS93]. We currently have a working prototype of SSAPRE that includes strength reduction and linear function test replacement.

Processing expressions one at a time also allows other possibilities for SSAPRE by customizing the handling of different types of expressions. For example, one might suppress PRE for expressions that are branch conditions because the branch instructions can evaluate the conditions without extra cost. One might also move selected loop-invariant operations out of loops to points that are not down-safe because they will not raise exceptions. Since SSAPRE works bottom-up with respect to an expression tree, it can reassociate the expression tree when no optimization opportunity was found with the original form. This last possibility represents a different approach for addressing the code shape issue in PRE discussed in [BC94]. We intend to report on any interesting results in future publications.

Acknowledgement

The authors would like to thank Ash Munshi, Ron Price and Ross Towle for their support of this work in the MIPSpro compilers. Peter Dahl and Mark Streich also contributed to the work described in this paper. Lastly we thank the conference referees, whose comments helped improve this paper.

References

[AWZ88] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of values in programs. In *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages*, pages 1–11, January 1988.

[BC94] P. Briggs and K. Cooper. Effective partial redundancy elimination. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 159–170, June 1994.

[CCF91] J. Choi, R. Cytron, and J. Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Conference Record of the Eighteenth ACM Symposium on Principles of Programming Languages*, pages 55–66, January 1991.

[CCL+96] F. Chow, S. Chan, S. Liu, R. Lo, and M. Streich. Effective representation of aliases and indirect memory operations in ssa form. In *Proceedings* *of the Sixth International Conference on Compiler Construction*, pages 253–267, April 1996.

[CFR+91] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Programming Languages and Systems*, 13(4):451–490, October 1991.

[CHKW86] F. Chow, M. Himelstein, E. Killian, and L. Weber. Engineering a risc compiler. In *Proceedings of IEEE COMPCON*, pages 132–137, March 1986.

[Cho83] F. Chow. A portable machine-independent global optimizer – design and measurements. Technical Report 83-254 (PhD Thesis), Computer Systems Laboratory, Stanford University, December 1983.

[Cho88] F. Chow. Minimizing register usage penalty at procedure calls. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 85–94, June 1988.

[Cli95] C. Click. Global code motion global value numbering. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 246–257, June 1995.

[CS95a] K. Cooper and T. Simpson. Scc-based value numbering. Technical Report CRPC-TR95636-S, Dept. of Computer Science, Rice University., October 1995.

[CS95b] K. Cooper and T. Simpson. Value-driven code motion. Technical Report CRPC-TR95637-S, Dept. of Computer Science, Rice University., October 1995.

[CSS96] J. Choi, V. Sarkar, and E. Schonberg. Incremental computation of static single assignment form. In *Proceedings of the Sixth International Conference on Compiler Construction*, pages 223–237, April 1996.

[Dha89] D. Dhamdhere. A new algorithm for composite hoisting and strength reduction optimization (+ corrigendum). *Journal of Computer Mathematics*, 27:1–14 (+ 31–32), 1989.

[DRZ92] D. Dhamdhere, B. Rosen, and K. Zadeck. How to analyze large programs efficiently and informatively. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 212–223, June 1992.

[DS93] K. Drechsler and M. Stadel. A variation of knoop, rüthing and steffen's lazy code motion. *SIGPLAN Notices*, 28(5):29–38, May 1993.

[GSW95] M. Gerlek, E. Stoltz, and M. Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven ssa form. *ACM Trans. on Programming Languages and Systems*, 17(1):85–122, January 1995.

[GWS94] M. Gerlek, M. Wolfe, and E. Stoltz. A reference chain approach for live variables. Technical Report CSE 94-029, Oregon Graduate Institute, April 1994.

[Joh94] R. Johnson. Efficient program analysis using dependence flow graphs. Technical Report (PhD Thesis), Dept. of Computer Science, Cornell University, August 1994.

[JPP94] R. Johnson, D. Pearson, and K. Pingali. The program structure tree: Computing control regions in linear time. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 171–185, June 1994.

[KRS92] J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 224–234, June 1992.

[KRS93] J. Knoop, O. Rüthing, and B. Steffen. Lazy strength reduction. *Journal of Programming Languages*, 1(1):71–91, March 1993.

[KRS94a] J. Knoop, O. Rüthing, and B. Steffen. Optimal code motion: Theory and practice. *ACM Trans. on Programming Languages and Systems*, 16(4):1117–1155, October 1994.

[KRS94b] J. Knoop, O. Rüthing, and B. Steffen. Partial dead code elimination. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 147–158, June 1994.

[LLC96] S. Liu, R. Lo, and F. Chow. Loop induction variable canonicalization in parallelizing compilers. In *Proceedings of the Fourth International Conference on Parallel Architectures and Compilation Techniques*, pages 228–237, October 1996.

[MR79] E. Morel and C. Renvoise. Global optimization by supression of partial redundancies. *Comm. ACM*, 22(2):96–103, February 1979.

[RWZ88] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages*, pages 12–27, January 1988.

[SG95] V. Sreedhar and G. Gao. A linear time algorithm for placing ϕ-nodes. In *Conference Record of the Eighteenth ACM Symposium on Principles of Programming Languages*, pages 62–73, January 1995.

[SKL88] B. Schwarz, W. Kirchgässner, and R. Landwehr. An optimizer for Ada – Design, experiences and results. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 175–184, June 1988.

[Wol96] M. Wolfe. *High Performance Compilers For Parallel Computing*. Addison Wesley, 1996.

[WZ91] M. Wegman and K. Zadeck. Constant propagation with conditional branches. *ACM Trans. on Programming Languages and Systems*, 13(2):181–210, April 1991.