Chapter 11
Redundancy Elimination

Fred Chow

Redundancy elimination is an important category of optimizations performed by modern optimizing compilers. In the course of program execution, certain computations may be repeated multiple times and yield the same results. Such redundant computations can be eliminated by saving and reusing the results of the earlier computations instead of recomputing them later.

There are two types of redundancies: *full* redundancy and *partial* redundancy. A computation is fully redundant if the computation has occurred earlier regardless of the flow of control. The elimination of full redundancy is also called common subexpression elimination. A computation is partially redundant if the computation has occurred only along certain paths. Full redundancy can be regarded as a special case of partial redundancy where the redundant computation occurs regardless of the path taken.

There are two different views of a computation related to redundancy: how it is computed and the computed value. The former relates to the operator and the operands it operates on, which translates to how it is represented in the program representation. The latter refers to the value generated by the computation in the static sense.[1] As a result, algorithms for finding and eliminating redundancies can be classified into those that are *syntax-driven* and those that are *value-driven*. In syntax-driven analyses, two computations are the same if they are the same operation applied to the same operands that are program variables or constants. In this case, redundancy can arise only if the variables' values have not changed

[1] All values referred to in this chapter are static values viewed with respect to the program code. A static value can map to different dynamic values during program execution.

F. Chow (✉)
Huawei, Fremont, CA, USA
e-mail: fchow99@comcast.net

F. Rastello, F. Bouchez Tichadou (eds.), *SSA-based Compiler Design*,
https://doi.org/10.1007/978-3-030-80515-9_11

between the occurrences of the computation. In value-based analyses, redundancy arises whenever two computations yield the same value. For example, $a + b$ and $a + c$ compute the same result if b and c can be determined to hold the same value. In this chapter, we deal mostly with syntax-driven redundancy elimination. The last section will extend our discussion to value-based redundancy elimination.

In our discussion on syntax-driven redundancy elimination, our algorithm will focus on the optimization of a lexically identical expression, like $a + b$, that appears in the program. During compilation, the compiler will repeat the redundancy elimination algorithm on all the other lexically identified expressions in the program.

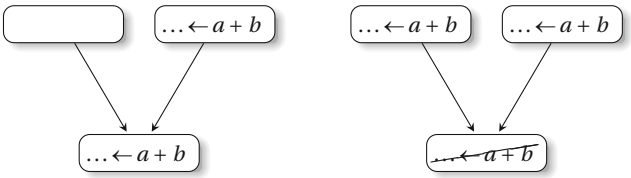
The style of the program representation can impact the effectiveness of the algorithm applied. We distinguish between *statements* and *expressions*. Expressions compute to values without generating any side effect. Statements have side effects as they potentially alter memory contents or control flow, and are not candidates for redundancy elimination. In dealing with lexically identified expressions, we advocate a maximal expression tree form of program representation. In this style, a large expression tree such as $a + b * c - d$ is represented as is without having to specify any assignments to temporaries for storing the intermediate values of the computation.[2] We also assume the Conventional SSA Form of program representation, in which each ϕ-web (see Chap. 2) is interference-free and the live ranges of the SSA versions of each variable do not overlap. We further assume the HSSA (see Chap. 16) form that completely models the aliasing in the program.

11.1 Why Partial Redundancy Elimination and SSA Are Related

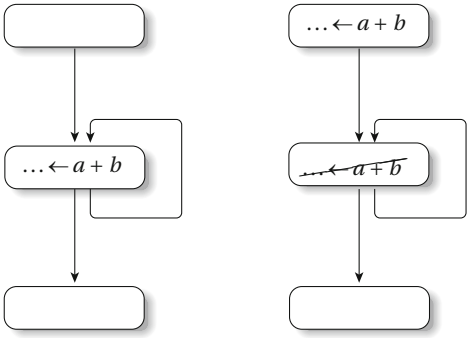
Figure 11.1 shows the two most basic forms of partial redundancy. In Fig. 11.1a, $a + b$ is redundant when the right path is taken. In Fig. 11.1b, $a + b$ is redundant whenever the back edge (see Sect. 4.4.1) of the loop is taken. Both are examples of *strictly* partial redundancies , in which insertions are required to eliminate the redundancies. In contrast, a full redundancy can be deleted without requiring any insertion. *Partial redundancy elimination (PRE)* is powerful because it subsumes global common subexpressions and loop-invariant code motion.

We can visualize the impact on redundancies of a single computation, as shown in Fig. 11.2. In the region of the control-flow graph dominated by the occurrence of $a + b$, any further occurrence of $a + b$ is fully redundant, assuming a and b are not modified. Following the program flow, once we are past the dominance frontiers, any further occurrence of $a + b$ is partially redundant. In constructing SSA form, dominance frontiers are where ϕs are inserted. Since partial redundancies start at dominance frontiers, partial redundancy elimination should borrow techniques

[2] The opposite of maximal expression tree form is the triplet form in which each arithmetic operation always defines a temporary.



(a) First example program, before and after PRE



(b) Second example program, before and after PRE

Fig. 11.1 Two basic examples of partial redundancy elimination

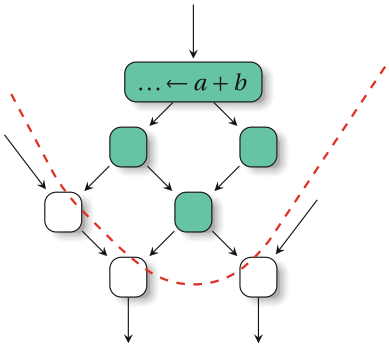
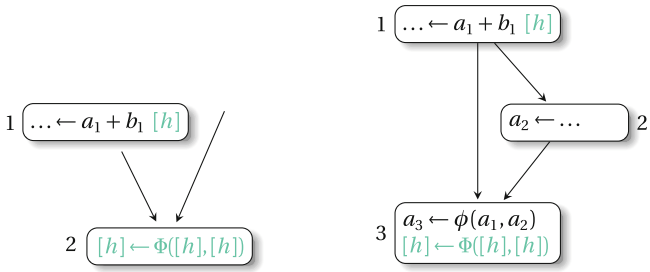


Fig. 11.2 Dominance frontiers (dashed) are boundaries between fully (highlighted basic blocks) and partially (normal basic blocks) redundant regions

from SSA ϕs insertion. In fact, the same sparse approach to modelling the use-def relationships among the occurrences of a program variable can be used to model the redundancy relationships among the different occurrences of $a + b$.

The algorithm that we present, named SSAPRE, performs PRE efficiently by taking advantage of the use-def information inherent in its input Conventional SSA Form. If an occurrence $a_j + b_j$ is redundant with respect to $a_i + b_i$, SSAPRE



(a) DF of expression occurrences **(b)** Expression alteration

Fig. 11.3 Examples of Φ-insertion

builds a redundancy edge that connects $a_i + b_i$ to $a_j + b_j$. To expose potential partial redundancies, we introduce the operator Φ at the dominance frontiers of the occurrences, which has the effect of factoring the redundancy edges at merge points in the control-flow graph.[3] The resulting *factored redundancy graph* (FRG) can be regarded as the SSA form for expressions.

To make the *expression SSA form* more intuitive, we introduce the hypothetical temporary h, which can be thought of as the temporary that will be used to store the value of the expression. The FRG can be viewed as the SSA graph for h. Observe that we have not yet determined where h should be defined or used. In referring to the FRG, a *use* node will refer to a node in the FRG that is not a definition.

The SSA form for h is constructed in two steps similar to ordinary SSA form: the Φ-insertion step followed by the Renaming step. In the Φ-insertion step, we insert Φs at the dominance frontiers of all the expression occurrences, to ensure that we do not miss any possible placement positions for the purpose of PRE, as in Fig. 11.3a. We also insert Φs caused by expression alteration. Such Φs are triggered by the occurrence of ϕs for any of the operands in the expression. In Fig. 11.3b, the Φ at block 3 is caused by the ϕ for a in the same block, which in turns reflects the assignment to a in block 2.

The Renaming step assigns SSA versions to h such that occurrences renamed to identical h-versions will compute to the same values. We conduct a pre-order traversal of the dominator tree similar to the Renaming step in SSA construction for variables, but with the following modifications: (1) In addition to a renaming stack for each variable, we maintain a renaming stack for the expression; (2) Entries on the expression stack are popped as our dominator tree traversal backtracks past the blocks where the expression originally received the version. 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.

[3] Adhering to the SSAPRE convention, we use lower case ϕs in the SSA form of variables and upper case Φs in the SSA form for expressions.

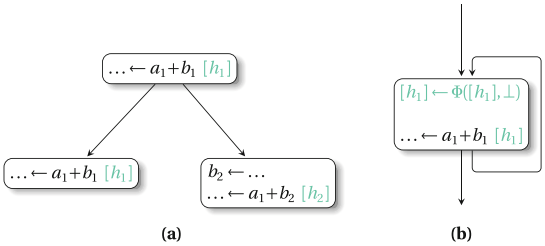


Fig. 11.4 Examples of expression renaming

There are three kinds of occurrences of the expression in the program: (real) the occurrences in the original program, which we call *real* occurrences; (Φ-def) the inserted Φs; and (Φ-use) the use operands of the Φs, which are regarded as occurring at the ends of the direct predecessor blocks of their corresponding edges. During the visitation in Renaming, a Φ is always given a new version. For a non-Φ, i.e., cases (real) and (Φ-use), we check the current version of every variable in the expression (the version on the top of each variable's renaming stack) against the version of the corresponding variable in the occurrence on the top of the expression's renaming stack. If all the variable versions match, we assign it the same version as the top of the expression's renaming stack. If one of the variable versions does not match, for case (real), we assign it a new version, as in the example of Fig. 11.4a; for case (Φ-use), we assign the special class ⊥ to the Φ-use to denote that the value of the expression is unavailable at that point, as in the example of Fig. 11.4b. If a new version is assigned, we push the version on the expression stack.

The FRG captures all the redundancies of $a+b$ in the program. In fact, it contains just the right amount of information for determining the optimal code placement. Because strictly partial redundancies can only occur at the Φ-nodes, insertions for PRE only need to be considered at the Φs.

11.2 How SSAPRE Works

Referring to the expression being optimized as X, we use the term *placement* to denote the set of points in the *optimized* program where X's computation occurs. In contrast, the *original computation points* refer to the points in the *original* program where X's computation took place. The *original* program will be transformed to the *optimized* program by performing a set of *insertions* and *deletions*.

The objective of SSAPRE is to find a placement that satisfies the following four criteria, in this order:

- **Correctness** : X is fully available at all the original computation points.
- **Safety**: There is no insertion of X on any path that did not originally contain X.
- **Computational optimality** : No other safe and correct placement can result in fewer computations of X on any path from entry to exit in the program.
- **Lifetime optimality** : Subject to computational optimality, the life range of the temporary introduced to store X is minimized.

Each occurrence of X at its original computation point can be qualified with exactly one of the following attributes: (1) *fully redundant*; (2) *strictly partially redundant*; (3) *non-redundant*.

As a code placement problem, SSAPRE follows the same two-step process used in all PRE algorithms. The first step determines the best set of insertion points that render fully redundant as many strictly partially redundant occurrences as possible. The second step deletes fully redundant computations, taking into account the effects of the inserted computations. As we consider this second step to be well understood, the challenge lies in the first step for coming up with the best set of insertion points. The first step will tackle the safety, computational optimality, and lifetime optimality criteria, while the correctness criterion is delegated to the second step. For the rest of this section, we only focus on the first step for finding the best insertion points, which is driven by the strictly partially redundant occurrences.

We assume that all critical edges in the control-flow graph have been removed by inserting empty basic blocks at such edges (see Algorithm 3.5). In the SSAPRE approach, insertions are only performed at Φ-uses. When we say a Φ is a candidate for insertion, it means we will consider insertions at its use operands to render X available at the entry to the basic block containing that Φ. An insertion at a Φ-use means inserting X at the incoming edge corresponding to that Φ operand. In reality, the actual insertion is done at the end of the direct predecessor block.

11.2.1 The Safety Criterion

As we have pointed out at the end of Sect. 11.1, insertions only need to be considered at the Φs. The safety criterion implies that we should only insert at Φs where X is *downsafe* (fully anticipated). Thus, we perform data-flow analysis on the FRG to determine the *downsafe* attribute for Φs. Data-flow analysis can be performed with linear complexity on SSA graphs, which we illustrate with the Downsafety computation.

A Φ is not *downsafe* if there is a control-flow path from that Φ along which the expression is not computed before program exit or before being altered by the redefinition of one of its variables. Except for loops with no exit, this can only happen in one of the following cases: (dead) there is a path to exit or an alteration of the expression along which the Φ result version is not used; or (transitive) the

Φ result version appears as the operand of another Φ that is not *downsafe*. Case (dead) represents the initialization for our backward propagation of ¬*downsafe*; all other Φs are initially marked *downsafe*. The Downsafety propagation is based on case (transitive). Since a real occurrence of the expression blocks the case (transitive) propagation, we define a *has_real_use* flag attached to each Φ operand and set this flag to true when the Φ operand is defined by another Φ and the path from its defining Φ to its appearance as a Φ operand crosses a real occurrence. The propagation of ¬*downsafe* is blocked whenever the *has_real_use* flag is true. Figure 11.1 gives the Downsafety propagation algorithm. The initialization of the *has_real_use* flags is performed in the earlier Renaming phase.

Algorithm 11.1: Downsafety propagation

```
1  foreach f ∈ {Φs in the program} do
2      if ∃ path P to program exit or alteration of expression along which f is not used then
         downsafe(f) ← false
3
4  foreach f ∈ {Φs in the program} do
5      if not downsafe(f) then
6          foreach operand ω of f do
7              if not has_real_use (ω) then Reset_downsafe(ω)

8  Function Reset_downsafe(X)
9      if def(X) is not a Φ then return
10     f ← def(X)
11     if not downsafe(f) then return
12     downsafe(f) ← false
13     foreach operand ω of f do
14         if not has_real_use (ω) then Reset_downsafe(ω)
```

11.2.2 The Computational Optimality Criterion

At this point, we have eliminated the unsafe Φs based on the safety criterion. Next, we want to identify all the Φs that are possible candidates for insertion, by disqualifying Φs that cannot be insertion candidates in any computationally optimal placement. An unsafe Φ can still be an insertion candidate if the expression is fully available there, though the inserted computation will itself be fully redundant. We define the *can_be_avail* attribute for the current step, whose purpose is to identify the region where, after appropriate insertions, the computation can become fully available. A Φ is ¬*can_be_avail* if and only if inserting there violates computational optimality. The *can_be_avail* attribute can be viewed as

$$can\_be\_avail(\Phi) = downsafe(\Phi) \cup avail(\Phi).$$

We could compute the *avail* attribute separately using the full availability analysis, which involves propagation in the forward direction with respect to the control-flow graph. But this would have performed some useless computation because we do not need to know its values within the region where the Φs are *downsafe*. Thus, we choose to compute *can_be_avail* directly by initializing a Φ to be ¬*can_be_avail* if the Φ is not *downsafe* and one of its operands is ⊥. In the propagation phase, we propagate ¬*can_be_avail* forward when a ¬*downsafe* Φ has an operand that is defined by a ¬*can_be_avail* Φ and that operand is not marked *has_real_use*.

After *can_be_avail* has been computed, computational optimality could be fulfiled simply by performing insertions at all the *can_be_avail* Φs. In this case, full redundancies would be created among the insertions themselves, but the subsequent full redundancy elimination step would remove any fully redundant inserted or non-inserted computation. This would leave the earliest computations as the optimal code placement.

11.2.3 The Lifetime Optimality Criterion

To fulfil lifetime optimality, we perform a second forward propagation called Later that is derived from the well-understood partial availability analysis. The purpose is to disqualify *can_be_avail* Φs where the computation is partially available based on the original occurrences of X. A Φ is marked *later* if it is not necessary to insert there because a later insertion is possible. In other words, there exists a computationally optimal placement under which X is not available immediately after the Φ. We optimistically consider all the *can_be_avail* Φs to be *later*, except in the following cases: (real) the Φ has an operand defined by a real computation; or (transitive) the Φ has an operand that is *can_be_avail* Φ marked not *later*. Case (real) represents the initialization for our forward propagation of not *later*; all other *can_be_avail* Φs are marked *later*. The Later propagation is based on case (transitive).

The final criterion for performing insertion is to insert at the Φs where *can_be_avail* and ¬*later* hold. We call such Φs *will_be_avail*. At these Φs, insertion is performed at each operand that satisfies either of the following conditions:

- it is ⊥; or
- *has_real_use* is false and it is defined by a ¬*will_be_avail* Φ

We illustrate our discussion in this section with the example of Fig. 11.5, where the program exhibits partial redundancy that cannot be removed by safe code motion. The two Φs with their computed data-flow attributes are as shown. If insertions were based on *can_be_avail*, $a + b$ would have been inserted at the exits of blocks 4 and 5 due to the Φ in block 6, which would have resulted in

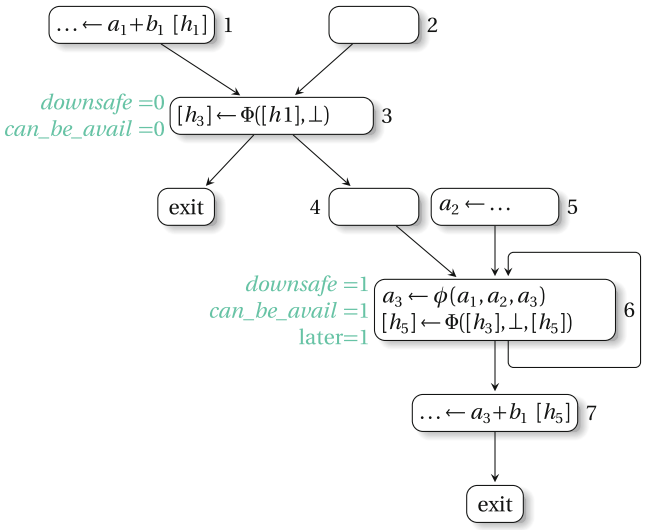


Fig. 11.5 Example to show the need for the *later* attribute

unnecessary code motion increasing register pressure. By considering *later*, no insertion is performed, which is optimal under safe PRE for this example.

11.3 Speculative PRE

If we ignore the safety requirement of PRE discussed in Sect. 11.2, the resulting code motion will involve speculation. Speculative code motion suppresses redundancy in some paths at the expense of another path where the computation is added but result is unused. As long as the paths that are burdened with more computations are executed less frequently than the paths where the redundant computations are avoided, a net gain in program performance can be achieved. Thus, speculative code motion should only be performed when there are clues about the relative execution frequencies of the paths involved.

Without profile data, speculative PRE can be conservatively performed by restricting it to loop-invariant computations. Figure 11.6 shows a loop-invariant computation $a + b$ that occurs in a branch inside the loop. This loop-invariant code motion is speculative because, depending on the branch condition inside the loop, it may be executed zero times, while moving it to the loop header causes it to execute once. This speculative loop-invariant code motion is profitable unless the path inside the loop containing the expression is never taken, which is usually not the case. When performing SSAPRE, marking Φs located at the start of loop bodies as downsafe will effect speculative loop-invariant code motion.

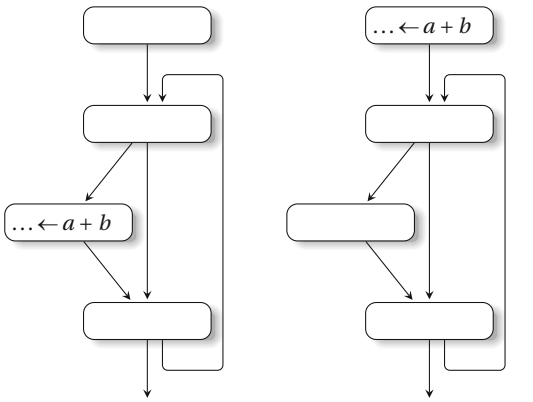


Fig. 11.6 Speculative loop-invariant code motion

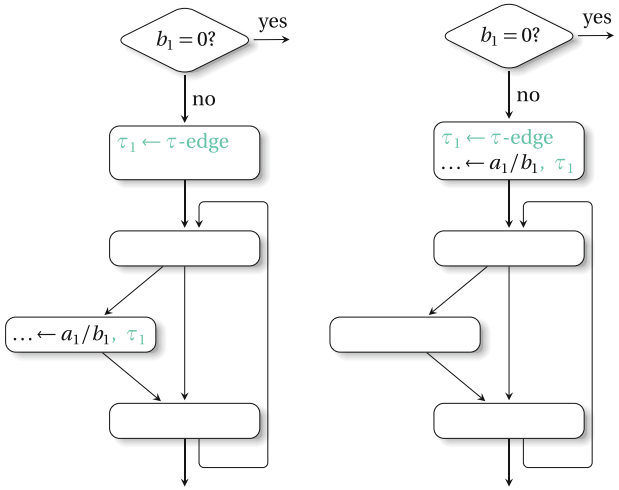
(a) Before PRE **(b)** After speculative PRE

Computations such as indirect loads and divides are called *dangerous* computations because they may generate a fault. Dangerous computations in general should not be speculated. As an example, if we replace the expression $a + b$ in Fig. 11.6 by a/b and the speculative code motion is performed, it may cause a runtime divide-by-zero fault after the speculation because b can be 0 at the loop header, while it is never 0 in the branch that contains a/b inside the loop body.

Dangerous computations are sometimes protected by tests (or guards) placed in the code by the programmers or automatically generated by language compilers such as those for Java. When such a test occurs in the program, we say the dangerous computation is *safety-dependent* on the control-flow point that establishes its safety. At the points in the program where its safety dependence is satisfied, the dangerous instruction is *fault-safe* and can still be speculated.

We can represent safety dependencies as value dependencies in the form of abstract τ variables. Each successful runtime test defines a τ variable on its fall-through path. During SSAPRE, we attach these τ variables as additional operands to the dangerous computations related to the test. The τ variables are also put into SSA form, so their definitions can be found by following the use-def chains. The definitions of the τ variables have abstract right-hand-side values that are not allowed to be involved in any optimization. Because they are abstract, they are also omitted in the generated code after the SSAPRE phase. A dangerous computation can be defined to have more than one τ operand, depending on its semantics. When all its τ operands have definitions, it means the computation is fault-safe; otherwise, it is unsafe to speculate. By taking the τ operands into consideration, speculative PRE automatically honors the fault-safety of dangerous computations when it performs speculative code motion.

In Fig. 11.7, the program contains a non-zero test for b. We define an additional τ operand for the divide operation in a/b in SSAPRE to provide the information about whether a non-zero test for b is available. At the start of the region guarded by the non-zero test for b, the compiler inserts the definition of τ_1 with the abstract



(a) Before PRE **(b)** After speculative PRE

Fig. 11.7 Speculative and fault-safe loop-invariant code motion

right-hand-side value τ-edge. Any appearance of a/b in the region guarded by the non-zero test for b will have τ_1 as its τ operand. Having a defined τ operand allows a/b to be freely speculated in the region guarded by the non-zero test, while the definition of τ_1 prevents any hoisting of a/b past the non-zero test.

11.4 Register Promotion via PRE

Variables and most data in programs normally start out residing in memory. It is the compiler's job to promote those memory contents to registers as much as possible to speed up program execution. Load and store instructions have to be generated to transfer contents between memory locations and registers. The compiler also has to deal with the limited number of physical registers and find an allocation that makes the best use of them. Instead of solving these problems all at once, we can tackle them as two smaller problems separately:

1. Register promotion—We assume there is an unlimited number of registers, called *pseudo-registers* (also called symbolic registers, virtual registers, or temporaries). Register promotion will allocate variables to pseudo-registers whenever possible and optimize the placement of the loads and stores that transfer their values between memory and registers.
2. Register allocation (see Chap. 22)—This phase will fit the unlimited number of pseudo-registers to the limited number of *real* or *physical* registers.

In this chapter, we only address the register promotion problem because it can be cast as a redundancy elimination problem.

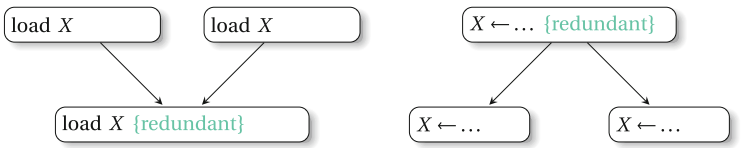
11.4.1 Register Promotion as Placement Optimization

Variables with no aliases are trivial register promotion candidates. They include the temporaries generated during PRE to hold the values of redundant computations. Variables in the program can also be determined via compiler analysis or by language rules to be alias-free. For these trivial candidates, one can rename them to unique pseudo-registers, and no load or store needs to be generated.

Our register promotion is mainly concerned with scalar variables that have aliases, indirectly accessed memory locations and constants. A scalar variable can have aliases whenever its address is taken, or if it is a global variable, since it can be accessed by function calls. A constant value is a register promotion candidate whenever some operations using it have to refer to it through register operands.

Since the goal of register promotion is to obtain the most efficient placement for loads and stores, register promotion can be modelled as two separate problems: PRE of loads, followed by PRE of stores. In the case of constant values, our use of the term *load* will extend to referring to the operation performed to put the constant value in a register. The PRE of stores does not apply to constants.

From the point of view of redundancy, loads behave like expressions: the later occurrences are the ones to be deleted. For stores, the reverse is true: as illustrated in the examples of Fig. 11.8, the earlier stores are the ones to be deleted. The PRE of stores, also called *partial dead code elimination*, can thus be treated as the dual of the PRE of loads. Thus, performing PRE of stores has the effects of moving stores forward while inserting them as early as possible. Combining the effects of the PRE of loads and stores results in optimal placements of loads and stores while minimizing the live ranges of the pseudo-registers, by virtue of the computational and lifetime optimality of our PRE algorithm.



(a) redundant load **(b)** redundant store

Fig. 11.8 Duality between load and store redundancies

11.4.2 Load Placement Optimization

PRE applies to any computation, including loads from memory locations or creation of constants. In program representations, loads can be either indirect through a pointer or direct. Indirect loads are automatically covered by the PRE of expressions. Direct loads correspond to scalar variables in the program, and since our input program representation is in HSSA form, the aliasing that affects the scalar variables is completely modelled by the χ and μ functions. In our representation, both direct loads and constants are leaves of the expression trees. When we apply SSAPRE to direct loads, since the hypothetical temporary h can be regarded as the candidate variable itself, the FRG corresponds somewhat to the variable's SSA graph, so the Φ-insertion step and Rename step can be streamlined.

When working on the PRE of memory loads, it is important to also take into account the stores, which we call *l-value* occurrences. A store of the form $X \leftarrow$ $<$ `expr` $>$ can be regarded as being made up of the sequence:

$$r \leftarrow < \texttt{expr} >$$
$$X \leftarrow r$$

Because the pseudo-register r contains the current value of X, any subsequent occurrences of the load of X can reuse the value from r and thus can be regarded as redundant. Figure 11.9 gives examples of loads made redundant by stores.

When we perform the PRE of loads, we thus take the store occurrences into consideration. The Φ-insertion step will insert Φs at the iterated dominance frontiers of store occurrences. In the Rename step, a store occurrence is always given a new h-version, because a store is a definition. Any subsequent load renamed to the same h-version is redundant with respect to the store.

We apply the PRE of loads (LPRE) first, followed by the PRE of stores (STRE). This ordering is based on the fact that LPRE is not affected by the result of STRE, but LPRE creates more opportunities for the SPRE by deleting loads that would otherwise have blocked the movement of stores. In addition, speculation is required for the PRE of loads and stores in order for register promotion to do a decent job in loops.

The example in Fig. 11.10 illustrates what is discussed in this section. During LPRE, $A \leftarrow \ldots$ is regarded as a store occurrence. The hoisting of the load of A to

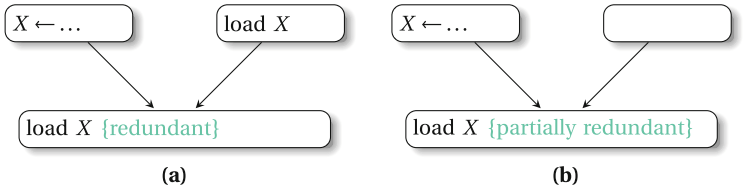
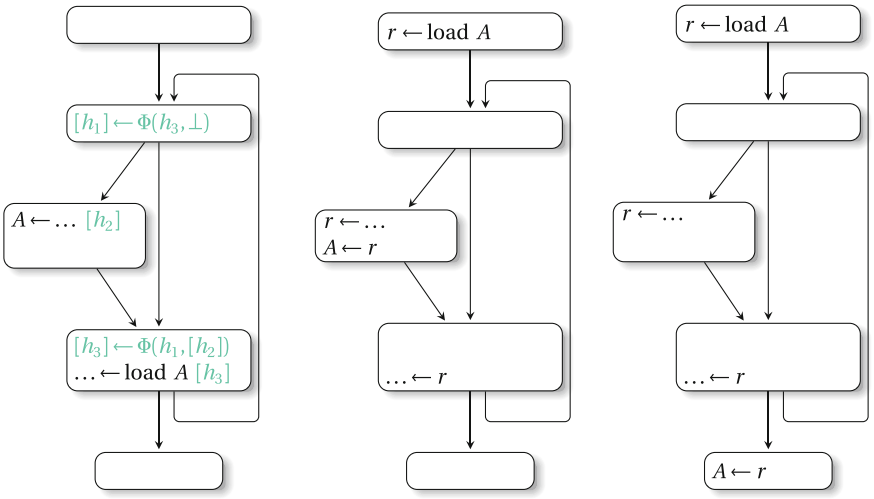


Fig. 11.9 Redundant loads after stores



(a) original **(b)** after LPRE **(c)** followed by SPRE

Fig. 11.10 Register promotion via load PRE followed by store PRE

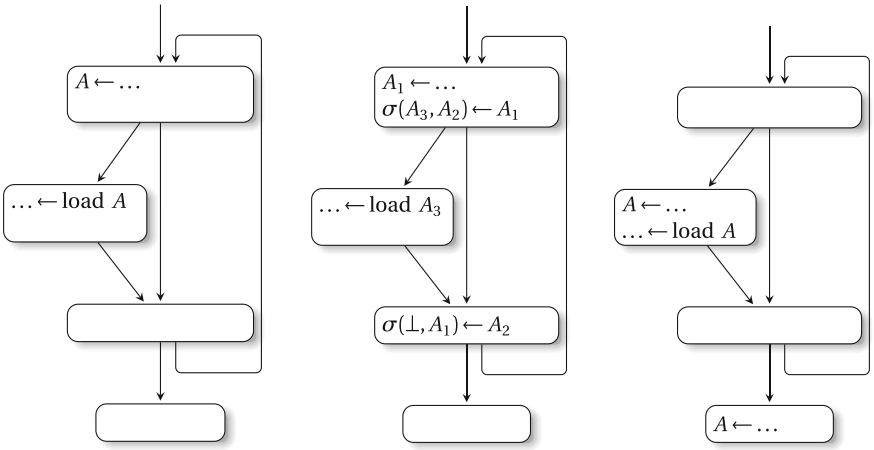
the loop header does not involve speculation. The occurrence of $A \leftarrow \ldots$ causes r to be updated by splitting the store into the two statements $r \leftarrow \ldots$; $A \leftarrow r$. In the PRE of stores (SPRE), speculation is needed to sink $A \leftarrow \ldots$ to outside the loop because the store occurs in a branch inside the loop. Without performing LPRE first, the load of A inside the loop would have blocked the sinking of $A \leftarrow \ldots$.

11.4.3 Store Placement Optimization

As mentioned earlier, SPRE is the dual of LPRE. Code motion in SPRE will have the effect of moving stores forward with respect to the control-flow graph. Any presence of (aliased) loads has the effect of blocking the movement of stores or rendering the earlier stores non-redundant.

To apply the dual of the SSAPRE algorithm, it is necessary to compute a program representation that is the dual of the SSA form, the *static single use* (SSU) form (see Chap. 13—SSU is a special case of SSI). In SSU, use-def edges are factored at divergence points in the control-flow graph using σ-functions (see Sect. 13.1.4). Each use of a variable establishes a new version (we say the load *uses* the version), and every store reaches exactly one load.

We call our store PRE algorithm SSUPRE, which is made up of the corresponding steps in SSAPRE. The insertion of σ-functions and renaming phases constructs the SSU form for the variable whose store is being optimized. The data-flow analyses consist of UpSafety to compute the *upsafe* (fully available)



(a) original **(b)** SSU form **(c)** after SPRE

Fig. 11.11 Example of program in SSU form and the result of applying SSUPRE

attribute, CanBeAnt to compute the *can_be_ant* attribute, and Earlier to compute the *earlier* attribute. Though store elimination itself does not require the introduction of temporaries, lifetime optimality still needs to be considered for the temporaries introduced in the LPRE phase, which hold the values to the point where the stores are placed. It is desirable not to sink the stores too far down.

Figure 11.11 gives the SSU form and the result of SSUPRE on an example program. The sinking of the store to outside the loop is traded for the insertion of a store in the branch inside the loop. The optimized code no longer exhibits any store redundancy.

11.5 Value-Based Redundancy Elimination

The PRE algorithm we have described so far is not capable of recognizing redundant computations among lexically different expressions that yield the same value. In this section, we discuss redundancy elimination based on value analysis.

11.5.1 Value Numbering

The term *value number* originates from a hash-based method for recognizing when two expressions evaluate to the same value within a basic block. The value number of an expression tree can be regarded as the index of its hashed entry in the hash table. An expression tree is hashed recursively bottom-up, starting with the leaf

$a \leftarrow \ldots$	$\mathsf{VN}(a) = v_1$
$c \leftarrow \ldots$	$\mathsf{VN}(c) = v_2$
$b \leftarrow c$	$\mathsf{VN}(b) = \mathsf{VN}(c) = v_2$
$c \leftarrow a + c$	$\mathsf{VN}(c) = \mathsf{VN}(+(\mathsf{VN}(a), \mathsf{VN}(c))) = \mathsf{VN}(+(v_1, v_2)) = v_3$
$d \leftarrow a + b$	$\mathsf{VN}(d) = \mathsf{VN}(+(\mathsf{VN}(a), \mathsf{VN}(b))) = \mathsf{VN}(+(v_1, v_2)) = v_3$

(a) straight-line code **(b)** value numbers

Fig. 11.12 Value numbering in a local scope

$a_1 \leftarrow \ldots$	$\mathsf{VN}(a_1) = v_1$
$c_1 \leftarrow \ldots$	$\mathsf{VN}(c_1) = v_2$
$b_1 \leftarrow c_1$	$\mathsf{VN}(b_1) = \mathsf{VN}(c_1) = v_2$
$c_2 \leftarrow a_1 + c_1$	$\mathsf{VN}(c_2) = \mathsf{VN}(+(\mathsf{VN}(a_1), \mathsf{VN}(c_1))) = \mathsf{VN}(+(v_1, v_2)) = v_3$
$d_1 \leftarrow a_1 + b_1$	$\mathsf{VN}(d_1) = \mathsf{VN}(+(\mathsf{VN}(a_1), \mathsf{VN}(b_1))) = \mathsf{VN}(+(v_1, v_2)) = v_3$

(a) processed statements **(b)** value numbers

Fig. 11.13 Global value numbering on SSA form

nodes. Each internal node is hashed based on its operator and the value numbers of its operands. The local algorithm for value numbering will conduct a scan down the instructions in a basic block, assigning value numbers to the expressions. At an assignment, the assigned variable will be assigned the value number of the right-hand side expression. The assignment will also cause any value number that refers to that variable to be killed. For example, the program code in Fig. 11.12a will result in the value numbers v_1, v_2, and v_3 shown in Fig. 11.12b. Note that variable c is involved with both value numbers v_2 and v_3 because it has been redefined.

SSA form enables value numbering to be easily extended to the global scope, called global value numbering (GVN), because each SSA version of a variable corresponds to at most one static value for the variable. In the example of Fig. 11.13, a traversal along any topological ordering of the SSA graph can be used to assign value numbers to variables. One subtlety is regarding the ϕ-functions. When we value number a ϕ-function, we would like the value numbers for its use operands to have been determined already. One strategy is to perform the global value numbering by visiting the nodes in the control-flow graph in a reverse post-order traversal of the dominator tree. This traversal strategy can minimize the instances when a ϕ-use has an unknown value number, which arises only in the case of back edges from loops. When this arises, we have no choice but to assign a new value number to the variable defined by the ϕ-function. For example, in the following loop:

```
1 i1 ← 0
2 j1 ← 0
3 while <cond> do
4 |   i2 ← φ(i3, i1)
5 |   j2 ← φ(j3, j1)
6 |   i3 ← i2 + 4
7 |   j3 ← j2 + 4
```

When we try to hash a value number for either of the two ϕs, the value numbers for i_3 and j_3 are not yet determined. As a result, we create different value numbers for i_2 and j_2. This makes the above algorithm unable to recognize that i_2 and j_2 can be given the same value number, or that i_3 and j_3 can be given the same value number.

The above hash-based value numbering algorithm can be regarded as pessimistic, because it will not assign the same value number to two different expressions unless it can prove they compute the same value. There exists a different approach (see Sect. 11.6 for references) to performing value numbering that is not hash-based and is optimistic. It does not depend on any traversal over the program's flow of control and so is not affected by the presence of back edges. The algorithm partitions all the expressions in the program into *congruence classes*. Expressions in the same congruence class are considered *equivalent* because they evaluate to the same static value. The algorithm is optimistic because when it starts, it assumes all expressions that have the same operator to be in the same congruence class. Given two expressions within the same congruence class, if their operands at the same operand position belong to different congruence classes, the two expressions may compute to different values and thus should not be in the same congruence class. This is the subdivision criterion. As the algorithm iterates, the congruence classes are subdivided into smaller ones, while the total number of congruence classes increases. The algorithm terminates when no more subdivisions can occur. At this point, the set of congruence classes in this final partition will represent all the values in the program that we care about, and each congruence class is assigned a unique value number.

While such a partition-based algorithm is not obstructed by the presence of back edges, it does have its own deficiencies. Because it has to consider one operand position at a time, it is not able to apply commutativity to detect more equivalences. Since it is not applied bottom-up with respect to the expression tree, it is not able to apply algebraic simplifications while value numbering. To get the best of both the hash-based and the partition-based algorithms, it is possible to apply the two algorithms independently and then combine their results together to shrink the final set of value numbers.

11.5.2 Redundancy Elimination Under Value Numbering

So far, we have discussed finding computations that compute to the same values but have not addressed eliminating the redundancies among them. Two computations that compute to the same value exhibit redundancy only if there is a control-flow path that leads from one to the other.

An obvious approach is to consider PRE for each value number separately. This can be done by introducing, for each value number, a temporary that stores the redundant computations. But value-number-based PRE has to deal with the issue of *how* to generate an insertion. Because the same value can come from different forms of expressions at different points in the program, it is necessary to determine which

form to use at each insertion point. If the insertion point is outside the live range of any variable version that can compute that value, then the insertion point has to be disqualified. Due to this complexity, and the expectation that strictly partial redundancy is rare among computations that yield the same value, it seems to be sufficient to perform only full redundancy elimination among computations that have the same value number.

However, it is possible to broaden the scope and consider PRE among lexically identical expressions and value numbers at the same time. In this hybrid approach, it is best to relax our restriction on the style of program representation described in Sect. 11. By not requiring Conventional SSA Form, we can more effectively represent the flow of values among the program variables. By considering the live range of each SSA version to extend from its definition to program exit, we allow its value to be used whenever convenient. The program representation can even be in the form of triplets, in which the result of every operation is immediately stored in a temporary. It will just assign the value number of the right-hand side to the left-hand-side variables. This hybrid approach (GVN-PRE—see below) can be implemented based on an adaptation of the SSAPRE framework. Since each ϕ-function in the input can be viewed as merging different value numbers from the direct predecessor blocks to form a new value number, the Φ-function insertion step will be driven by the presence of ϕs for the program variables. Several FRGs can be formed, each being regarded as a representation of the flow and merging of computed values. Using each individual FRG, PRE can be performed by applying the remaining steps of the SSAPRE algorithm.

11.6 Further Reading

The concept of partial redundancy was first introduced by Morel and Renvoise. In their seminal work [201], Morel and Renvoise showed that global common subexpressions and loop-invariant computations are special cases of partial redundancy, and they formulated PRE as a code placement problem. The PRE algorithm developed by Morel and Renvoise involves bidirectional data-flow analysis, which incurs more overhead than unidirectional data-flow analysis. In addition, their algorithm does not yield optimal results in certain situations. A better placement strategy, called lazy code motion (LCM), was later developed by Knoop et al. [170, 172]. It improved on Morel and Renvoise's results by avoiding unnecessary code movements, by removing the bidirectional nature of the original PRE data-flow analysis and by proving the optimality of their algorithm. Since lazy code motion was introduced, there have been alternative formulations of PRE algorithms that achieve the same optimal results but differ in the formulation approach and implementation details [98, 106, 217, 313].

The above approaches to PRE are all based on encoding program properties in bit-vector forms and the iterative solution of data-flow equations. Since the bit-vector representation uses basic blocks as its granularity, a separate algorithm is

needed to detect and suppress local common subexpressions. Chow et al. [73, 164] came up with the first SSA-based approach to perform PRE. Their SSAPRE algorithm is an adaptation of LCM that takes advantage of the use-def information inherent in SSA. It avoids having to encode data-flow information in bit-vector form and eliminates the need for a separate algorithm to suppress local common subexpressions. Their algorithm was the first to make use of SSA to solve data-flow problems for expressions in the program, taking advantage of SSA's sparse representation so that fewer steps are needed to propagate data-flow information. The SSAPRE algorithm thus brings the many desirable characteristics of SSA-based solution techniques to PRE.

In the area of speculative PRE, Murphy et al. [206] introduced the concept of fault-safety and used it in the SSAPRE framework for the speculation of dangerous computations. When execution profile data are available, it is possible to tailor the use of speculation to maximize runtime performance for the execution that matches the profile. Xue and Cai [312] presented a computationally and lifetime optimal algorithm for speculative PRE based on profile data. Their algorithm uses data-flow analysis based on bit-vector and applies minimum cut to flow networks formed out of the control-flow graph to find the optimal code placement. Zhou et al. [317] applied the minimum cut approach to flow networks formed out of the FRG in the SSAPRE framework to achieve the same computational and lifetime optimal code motion. They showed their sparse approach based on SSA results in smaller flow networks, enabling the optimal code placements to be computed more efficiently.

Lo et al. [187] showed that register promotion can be achieved by load placement optimization followed by store placement optimization. Other optimizations can potentially be implemented using the SSAPRE framework, for instance code hoisting, register shrink-wrapping [70], and live range shrinking. Moreover, PRE has traditionally provided the context for integrating additional optimizations into its framework. They include operator strength reduction [171] and linear function test replacement [163].

Hashed-based value numbering originated from Cocke and Schwartz [77], and Rosen et al. [249] extended it to global value numbering based on SSA. The partition-based algorithm was developed by Alpern et al. [6]. Briggs et al. [49] presented refinements to both the hash-based and partition-based algorithms, including applying the hash-based method in a post-order traversal of the dominator tree.

VanDrunen and Hosking proposed A-SSAPRE (anticipation-based SSAPRE) which removes the requirement of Conventional SSA Form and is best for program representations in the form of triplets [296]. Their algorithm determines optimization candidates and constructs FRGs via a depth-first, pre-order traversal over the basic blocks of the program. Within each FRG, non-lexically identical expressions are allowed, as long as there are potential redundancies among them. VanDrunen and Hosking [297] subsequently presented GVN-PRE (Value-based Partial Redundancy Elimination), which is claimed to subsume both PRE and GVN.