CHAPTER 1

Redundancy Elimination

F. Chow

Progress: 50%

Review in progress

Author: F. Chow

1.1 Introduction

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 that yield the same results. Such redundant computations can be eliminated by saving the results of the ~~non-redundant~~ computations for reuse at the redundant computations.

previous

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, and if applied at the global scope, it is called global 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 computation has occurred regardless of which path is taken.

You need to develop this a little bit further. What are the (dis)advantages of both methods? The lexical approach is criticized by Van Drunen although I think value numbering based PRE does not allow to optimize a code like that (while lexical can):

There are two different methods for deciding whether two computations are the same: the *lexical* method and the *semantic* method. Under the lexical method, two computations are the same if they are written the same way using variables and/or constants before converting to SSA, like $a + 3$. In this case, redundancy can arise only if the variables' values have not changed between the occurrences of the computation. Under the semantic method, two computations are the same if they are the same operation performed on operands that are not necessarily identical by name,

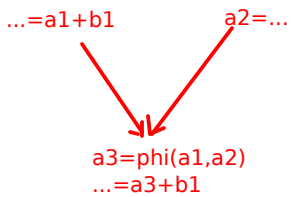


Fig 1

but known to have the same values. For example, $a + b$ and $a + c$ will compute the same result if b and c are known to hold the same value. In this chapter, we are mostly dealing with lexically identified expressions. Our algorithm discussion will focus on the optimization of an expression, like $a + b$, that appears in the program. The compiler will repeat the redundancy elimination algorithm on all the other lexically identified expressions in the program. Section 1.6 will discuss redundancy elimination among expressions that have been semantically proven to yield the same value.

The concept of partial redundancy was first introduced by Morel and Renvoise. Before the technique of partial redundancy elimination was developed, optimizing compilers have been performing global common subexpression elimination and loop invariant code motion in separate global optimization phases. In their seminal work [16], Morel and Renvoise showed that global common subexpressions and loop-invariant computations are special cases of partial redundancy that can be subsumed by the single optimization of partial redundancy elimination (PRE). Morel and Renvoise formulated PRE as a code placement problem, in which the best set of insertion points for the expression being optimized is to be determined. Such insertions render some original computations to be fully redundant, so they can be trivially deleted. The PRE algorithm developed by Morel and Renvoise involves bi-directional data flow analysis, which incurs more overhead than uni-directional data flow analysis. In addition, their algorithm does not yield optimal results in certain situations.

An better placement strategy, called lazy code motion (LCM), was later developed by Knoop *et al* [11][13]. It improved on Morel and Renvoise's results by avoiding unnecessary code movements and by removing the bi-directional nature of the original PRE data flow analysis. The code placement produced by lazy code motion is optimal: the number of computations during execution time cannot be further reduced by *safe* code motion [8], and the lifetimes of the temporaries introduced for storing the computed values are minimized. After 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[7][6][18][21].

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. An SSA-based approach to solve PRE was proposed by Chow *et al* [4][9]. Their SSAPRE algorithm is an adaptation of LCM to take 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 subexpressionsi. Their algorithm was 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 number of steps are needed to propagate data flow information. The SSAPRE algorithm thus brings the many desirable characteristics of SSA-based solution techniques to PRE, and further advance our understanding of this important optimization. Because SS-

Summaries all this to focus only on concepts that are useful to understand how your algorithm work. Remove _all_ references. You can put references in the last section that is dedicated to discussions, extentions, further readings. Notice that you can refer to a paper _only_ if it extends or adds anything to what is explained in this chapter. We do not care about previous work that is subsumed by this. Refer to Appel's book for the style I am waiting for.

APRE operates on a sparse representation, it precludes the need for any alternative formulation to its algorithm.

In this chapter, we only cover the conceptual bases of SSAPRE for the purpose of getting an intuitive understanding of its inner workings. The readers are referred to the original publications for the full algorithm[4][9].

=> last section

1.2 Why PRE and SSA are related

Figure 1.1 shows the two most basic forms of partial redundancy. In Figure 1.1(a), $a + b$ is redundant when the right path is taken. In Figure 1.1(b), $a + b$ is redundant whenever the branch-back edge 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.

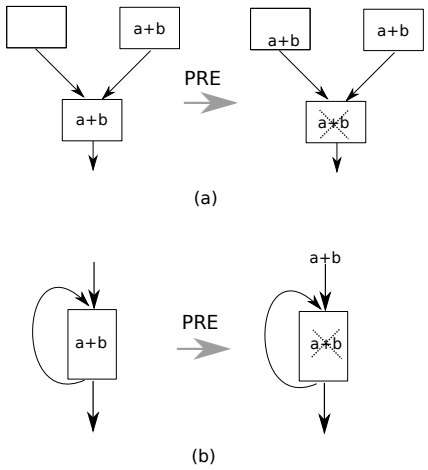


Fig. 1.1 Two basic examples of partial redundancy elimination.

We can visualize the impact on redundancies of a single computation as shown in Figure 1.2. 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 ϕ's are inserted. Since partial redundancies start at dominance frontiers, it must be related to SSA's ϕ's. In fact, the same sparse approach to modeling the use-def relationships among the occurrences of a program

variable can be used to model the redundancy relationships among the different occurrences of $a + b$.

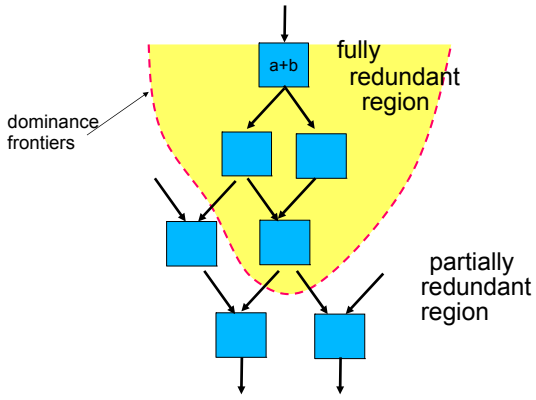


Fig. 1.2 Dominance frontiers are boundaries between fully and partially redundant regions.

It has to be conventional SSA. Please outline. This is because you address lexical method here. For value based this is not a requirement.

We assume the input program is already in SSA form. If an occurrence $a_j + b_j$ is redundant with respect to $a_i + b_i$, our representation has redundancy edges that connect $a_i + b_i$ 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.[1] 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 for reuse in order to suppress redundant computations. The constructed SSA form for h is not precise, because we have not yet determined where h should be defined or used.

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 Φ's 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 Figure 1.3(a). We also insert Φ's caused by expression alteration. Such Φ's are triggered by the occurrence of ϕ's for any of the operands in the expression. In

[1] Adhering to SSAPRE's convention, we use lower case ϕ's in the SSA form of variables and upper case Φ's in the SSA form for expressions.

For someone who considers value/semantic based redundancies (as SSA is highly appropriated for that) this would not make any sens (whereever ai and bi are available ai+bi can be computed so in this context expressions are not altered). So again this is important to outline the context of lexical method here as this might be very confusing otherwise.

Figure 1.3(b), 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.

name blocks in your figure

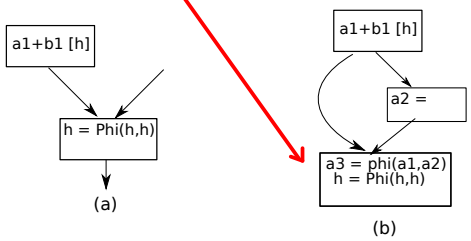


Fig. 1.3 Examples of Φ insertion

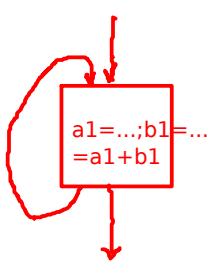
The Renaming step assigns SSA versions to h such that occurrences renamed to idential h-versions will compute to the same values. We conduct a pre-order traversal of the dominator tree similar to the renaming step in SSA construction for variables, but with the following modifications. In addition to a renaming stack for each variable, we maintain a renaming stack for the expression. 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.

But we _are_ under SSA so variables do not need to be "renamed". Maybe you should consider you initial code not to be under SSA. This would probably simplify the discussion (especially concerning expression alteration, lexical redundancy etc.)

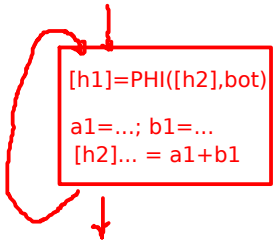
There are three kinds of occurrences of the expression in the program: (a) the occurrences in the original program, which we call *real* occurrences; (b) the Φ's inserted in the Φ-Insertion step; and (c) Φ operands, which are regarded as occurring at the ends of the predecessor blocks of their corresponding edges. During the visitation in Renaming, a Φ's is always given a new version. For a non-Φ, i.e., cases (a) and (c), 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 any of the variable versions does not match, for case (a), we assign it a new version, as in the example of Figure 1.4(a); for case (c), we assign the special class $\perp$ to the Φ operand to denote that the value of the expression is unavailable at that point, as in the example of Figure 1.4(b). 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 Φ's.

Actually not at the CFG node containing the PHI itself but at some of its predecessor nodes (you can add a footnote).



Initial SSA code



Corresponding FRG.

Fig 2

You have a new version h2 here (and this is what you want) because at the entry of the loop the "versions" of a & b are undef.
You would have a "use" of [h1] if the defs of a1 & b1 were before the loop.
This is an important subttle point. Here [h1] is (fortunately) not down safe.
Note that the "clean" way to handle it would have been
to avoid putting any PHI at the entry of the loop as a & b are not available here.

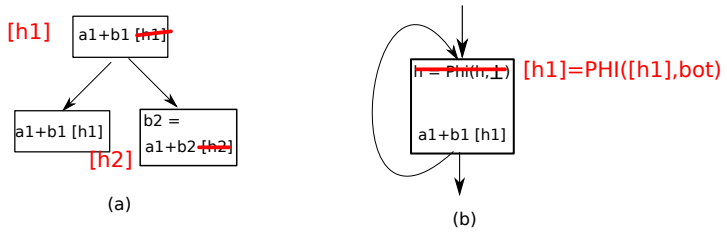


Fig. 1.4 Examples of expression renaming

1.3 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, *original computation points* refer to the points in the *original* program where X's computation took place. The objective of SSAPRE is to find a placement that satisfies the following four criteria in this order:

please insist that placement != insertion as the term is ambiguous

1. Correctness — X is fully available at all the original computation points.
2. Safety — There is no insertion of X on any path that did not originally contain X.
3. Computational optimality —No other safe & correct placement can result in fewer computations of X on any path in the program.
4. Lifetime optimality — Subject to the computational optimality, the life range of the temporary introduced to store X is minimized.

Any path from entry to exit?

Each occurrence of X at its original computation point can be qualified with exactly one of the following attributes:

- fully redundant
- strictly partially redundant (SPR)
- non-redundant

As a code placement problem, SSAPRE follows the same two-step process used in all PRE algorithsm. The first step determines the best set of insertion points that render as many SPR occurrences fully redundant as possible. The second step deletes fully redundant computations taking into account the effects of the inserted computations. Since the second full redundancy elimination step is trivial and 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 SPR occurrences.

refer to the SSA destruction chapter

We assume that all *critical edges* in the control flow graph have been removed by inserting empty basic blocks at such edges[2][XQ]. In the SSAPRE approach, insertions are only performed at Φ operands. When we say a Φ is a candidate for insertion, it means we will consider inserting at its operands to render X available at the entry to the basic block containing that Φ. An insertion at a Φ operand means inserting X at the incoming edge corresponding to that Φ operand. In reality, the actual insertion is done at the end of the predecessor block.

1.3.1 The Safety Criterion

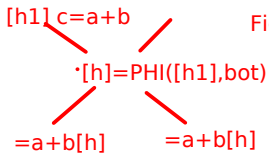
You _can_ insert were it is already fully available but you don't need to. Strictly speeking this sentence is not correct.

down safety is stronger than safety (= fully anticipated or fully available).

As we have pointed out at the end of Section 1.2, insertions only need to be considered at the Φ's. The safety criterion implies that we can only insert at Φ's where X is *downsafe* (fully anticipated). Thus, we perform data flow analysis on the FRG to determine the *downsafe* attribute for Φ's. ~~Data flow analysis can be performed with linear complexity on SSA graphs, which we illustrate with the Downsafety computation.~~

Some DF analysis are. But most are not. Here it is not linear in the size of the FRG as the initialization is linear in the CFG. ie same complexity than non-SSA version...

Use those notations in the figures:



For some people, anticipation concerns actual uses so as to avoid "waste". In this case "partial anticipation" being non dead code.
Here I understood that you talk about anticipability because a partially redundant exp is considered as a "use" of the previously refering one.
This notation is very confusing.

A Φ is not *downsafe* 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 or an alteration of the expression along which the Φ result version is not used; or (b) the Φ result version appears as the operand of another Φ that is not *downsafe*. Case (a) represents the initialization for our backward propagation of $\neg downsafe$; all other Φ's are initially marked $downsafe$. The Downsafety propagation is based on case (b). Since a real occurrence of the expression blocks the case (b) propagation, we mark 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 X to effect the blocking. Figure 1.5 gives the DownSafety propagation algorithm.

which path?...
has_real_use is set if the immediately dominating occurence is a real one.

You forgot to talk about the initialization of downsafe flag and has_real_use. This can be done during the rename phase.

1.3.2 The Computational Optimality Criterion

Can_be_avail is about marking the region of interest. It should be outlined that can_be_avail==safe

The best set of Φ's for performing insertion is arrived at via a process of elimination. At this point, we have eliminated the unsafe Φ's based on the safety criterion. We now seek to disqualify more Φ's from insertion consideration by identifying those that we can prove as violating the computational optimality criterion. This is done by performing the CanBeAvail propagation, which is in the forward direction.

The CanBeAvail propagation is derived from the well-understood full availability analysis. ~~We define a *Φ can_be_avail* if and only if inserting there will not violate~~

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

```
                                    procedure Reset_downsafe(X) {
1:      if (has_real_use(X) or def(X) is not a Φ)
2:          return
3:      f ← def(X)
4:      if (not downsafe(f))
5:          return
6:      downsafe(f) ← false
7:      for each operand ω of f do
8:          Reset_downsafe(ω)
                                    }
                                    procedure DownSafety {
9:      for each f ∈ {Φ's in the program} do
10:         if (not downsafe(f))
11:             for each operand ω of f do
12:                 Reset_downsafe(ω)
                                    }
```

Fig. 1.5 Algorithm for DownSafety

Please put the pseudo code for the initialization of has_real_use and downsafe. Insists in the fact that has_real_use is attached to the phi operand _not_ the corresponding def.

can_be_avail can be understood as the existence of a safe placement that makes it fully available.

~~computational optimality~~. ~~In~~ other words, a Φ is $\neg$*can_be_avail* if and only if inserting there violates computational optimality. This can happen only due to one of the following cases: (a) the Φ is not *downsafe* and one of its operands is $\perp$; or (b) the Φ is not *downsafe* and it has an operand that is a $\neg$*can_be_avail*Φ and that operand is not *has_real_use*. Case (a) represents the initialization for our forward propagation of $\neg$*can_be_avail*; all other Φ's are initially marked *can_be_avail*. The CanBeAvail propagation is based on case (b).

After *can_be_avail* has been computed, it is possible to perform insertions at all the *can_be_avail* Φ's. There would be full redundancies created among the insertions themselves, but they would not affect computational optimality because the subsequent full redundancy elimination step will remove any fully redundant inserted or non-inserted computation, leaving the earliest computations as the optimal code placement.[3]

1.3.3 The Lifetime Optimality Criterion

To fulfill 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* Φ's ~~that are 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. We optimistically regard all the *can_be_avail* Φ's to be *later*, except the following cases: (a) the Φ has an operand defined by a real computation; or (b) the Φ has an operand that is a Φ marked not *later*. Case (a) represents

confusing to talk about partial avail here.

can_be_avail

[3] This outcome is referred to as *busy code motion* ~~by~~ Knoop *et al.* to contrast with their lazy code motion.

the initialization for our forward propgation of not *later*; all other *can_be_avail* Φ's are marked *later*. The Later propagation is based on case (b).

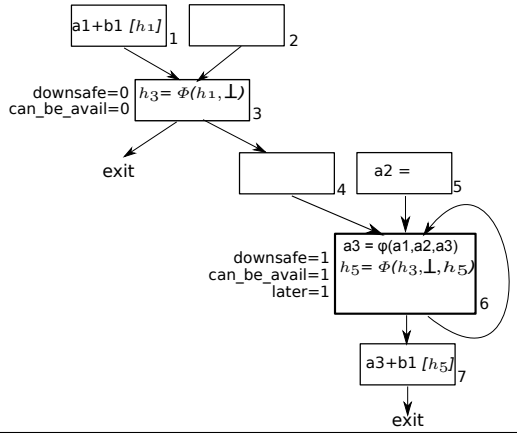


Fig. 1.6 Example to show the need of the *Later* attribute

rephrase

The final Φ's for performing insertion are the Φ's where *can_be_avail* and $\neg$*later* hold. We call such Φ's *will_be_avail*. At each of these Φ's, insertion is performed at each operand that satisfies either of the following condition:

emphasize

1. it is $\perp$; or
2. *has_real_use* is false and it is defined by a $\neg$*will_be_avail*Φ.

I think such an example is good to illustrate all the notations: has_real_use, bot, downsafe, can_be_avail, later

We illustrate our discussion in this section with the example of Figure 1.6, where the program exhibits partial redundancy that cannot be removed by safe code motion. The two Φ's with their computed data flow attributes are as shown. If insertions were based on *can_be_avail*, $a+b$ would have been inserted at blocks ~~4 and 5,~~ which would have resulted in unnecessary code motion that increases register pressure. By considering *later*, no insertion is performed, which is optimal under safe PRE for this example.

6 (there are no PHI in 4 & 5)

If a3 & b5 are not used in 7, code motion does reduce the register pressure here.

1.4 Speculative PRE

If we ignore the safety requirement of PRE discussed in Section 1.3, the resulting code motion will involve speculation. In the absence of systems support, speculation should only be applied to expressions that will not cause runtime exceptions or faults, since such exceptions or faults will alter the external behavior of the program. For example, indirect loads from unknown pointer values cannot be speculated unless the systems would mask out the effect of loading from invalid addresses. Speculative code motion suppresses redundancy in some path 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 1.7 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 time, while moving it to the loop header causes it to execute one time. 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 Φ's located at the start of loop bodies downsafe will effect speculative loop invariant code motion[14].

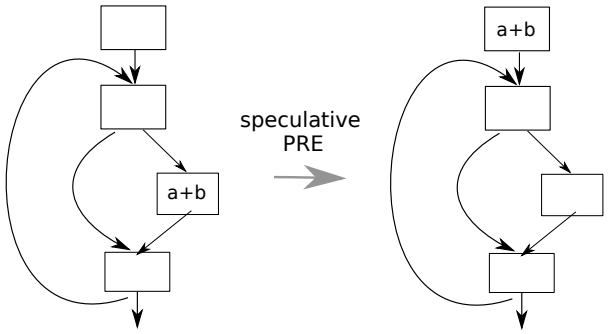


Fig. 1.7 Speculative loop-invariant code motion

To enable more code motion of unsafe operations, Murphy *et al.* modify SSAPRE to use the *fault-safety* property[17]. They define *dangerous* computations as operations that may fault but will otherwise have no observable side effects. Such computations include indirect loads and divides. Such dangerous computations are sometimes protected by tests (or guards) placed in the code by the programmers. Compilers for languages like Java also insert runtime checks to ensure faults never occur. When such a test occurs in the program, the dangerous computation is said to be *safety-dependent* on the control flow point that establishes its safety. A dangerous instruction is *fault-safe* at any point in the program where its safety dependence is satisfied.

no references in the main text. put it in last section "further readings".

Murphy *et al.* represent safety dependences as value dependences in the form of the abstract *tau values* described in [15]. Each check that succeeds will define a tau value on its fall-through path. During SSAPRE, dangerous computations will have additional *tau* operands attached to them. The tau operands are also variables

I should not have to read 15 to understand this

in SSA form, so their definitions can be found by following the use-def edges. The compiler inserts the definitions of the taus also with abstract right-hand-side values, like **tauedge**. Because they are abstract, they are omitted in the generated code after the SSAPRE phase. A dangerous computation can be defined to have more than one tau operands, depending on its semantics. When all its tau operands have definitions, it means the computation is fault-safe; otherwise, it is unsafe. By including the tau operands into consideration, speculative PRE automatically honors the fault-safety of dangerous computations when it performs speculative code motion.

?? what about other operands? They must also be defined

As an example, if we replace the expression $a+b$ in Figure 1.7 by a/b, the speculative code motion cannot be performed because if b is 0, the speculative insertion of a/b at the loop header will cause a run-time divide-by-zero fault. In Figure 1.8, the program contains a non-zero test for b. We define a tau operand for a/b in SSAPRE to provide the information whether a non-zero test for b is available. The presense of the non-zero test for b causes the compiler to insert the definition of $tau1$ with the abstract right-hand-side value **tauedge**. Since the divide inside the loop $a_1/b_1, tau_1$ has a tau operand whose definition is visible, speculative SSAPRE will force the Φ at the head of the loop body to be downsafe. When $a_1/b_1, tau_1$ is hoisted out of the loop, it automatically stops at the definition of tau_1 because PRE obeys value dependence.

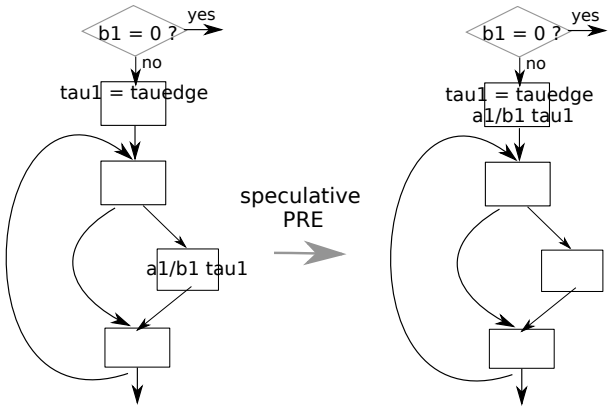


Fig. 1.8 Speculative and fault-safe loop-invariant code motion

Put all this in the last section of the chapter

When execution profile data are available, it is possible to tailor the use of speculation to maximize run-time performance for the execution that matches the profile. Xue and Cai presented a computationally and lifetime optimal algorithm for speculative PRE based on profile data[20]. Their algorithm uses bit-vector-based data flow analysis and applies minimum cut to flow networks formed out of the control flow graph to find the optimal code placement. Zhou et al. applies the minimum cut approach to flow networks formed out of the FRG in the SSAPRE framework to

I think that in general you should filter out for downsafety (or simply safety ie can_be_avail) PHI for which _one_ of the arguments (here a1 & b1 also) is not available.

achieve the same computational and lifetime optimal code motion[22]. They showed their sparse approach based on SSA results in smaller flow networks, enabling the optimal code placements to be computed more efficiently.

1.5 Register Promotion via PRE

Could you simplify this by just sticking on the optimization pb: you have data that are stored in memory and refered at different points in the program. Placing a load (resp. store) at each use (resp. def), might lead to redundant loads/stores as illustrated in Fig 1.9. Register promotion aims at optimizing the placement of loads/stores...

Register promotion refers to the important task in an optimizing compiler of identifying the data items that are candidates for register allocation in the program. To represent register allocation candidates, compilers commonly use an unlimited number of *pseudo-registers*. Pseudo-registers are also called symbolic registers or virtual registers, to distinguish them from real or physical registers. Pseudo-registers have no alias, and the process of assigning them to real registers involves only renaming them.

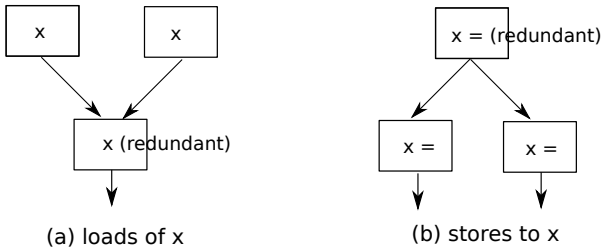
1.5.1 Register Promotion as Placement Optimization

Under PRE, the temporaries generated to hold the values of redundant computations are pseudo-registers. Local variables determined by the compiler to have no alias can be trivially renamed to pseudo-registers. All remaining register allocation candidates have to be assigned pseudo-registers through the process of register promotion. Register promotion is also responsible for generating the most efficient code to set up the data objects in pseudo-registers. Targets for register promotion include scalar variables, indirectly accessd memory locations and program constants. Load operations are needed to put them into pseudo-registers before they are used[4], and when there is an assignment, a store operation is needed. Since the goal of register promotion is to obtain the most efficient placements for the loads and stores, register promotion can be modeled as two separate problems: PRE of loads, followed by PRE of stores.

From the point of view of redundancy, loads are like expressions because the later occurrences are the ones to be deleted. For stores, the reverse is true: the earlier stores are the ones to be deleted, as is evident in the examples of Figure 1.9(a) and (b). The PRE of stores, also called *partial dead code elimination*, can thus be treated as the dual of the PRE of loads. Performing PRE of stores thus 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 optimalities of our PRE algorithm.

[4] Depending on the ISA, some constants may not need to be put in registers, and they should be excluded from register promotion.

For direct loads without aliasing, why isn't it the job of the register allocation to handle the placement of loads and stores?



Notations for loads & store here

Fig. 1.9 Duality between load and store redundancies

1.5.2 Load Placement Optimization

PRE applies to any computation including loads from memory locations and loads of constants. In program representations, loads can either be indirect through a pointer or direct. Direct loads and constants are leaves in expression trees. When we apply SSAPRE to direct loads, since the hypothetical temporary h can be regarded as the candidate variable itself, the Φ-insertion step and Rename step can be streamlined. In other words, the Φ's are the variable's ϕ's, and the h-versions are the variable's SSA version.

I think this is pointless and potentially confusing. Maybe you should outline that you restrict to accesses without aliasing here.

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:

Don't need this notation. Just talk about a store.

Notations

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

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 1.10 gives examples of loads made redundant by stores.

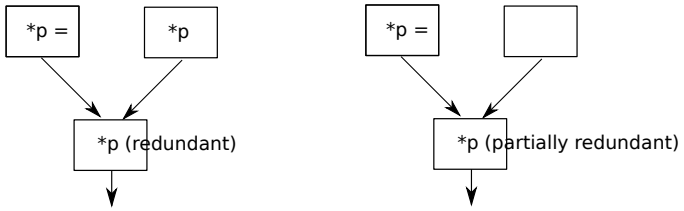


Fig. 1.10 Redundant loads after stores

When we perform the PRE of loads, we thus include the l-value occurrences into consideration. The Φ-insertion step will insert Φ's at the iterated dominance

frontiers of l-value occurrences. In the Rename step, an l-value 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 first, followed by the PRE of stores. This ordering is based on the fact that the PRE of loads is not affected by the results of the PRE of stores, but the PRE of loads creates more opportunities for the PRE of stores 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 Figure 1.11 illustrates what we discuss in this section. During the PRE of loads (LPRE), $a =$ is regarded as an l-value occurrence. The hoisting of the load of a to the loop header does not involve speculation. The occurrence of $a =$ causes r to be updated by splitting the store into the two statements $r =$ followed by $a = r$. In the PRE of stores (SPRE), speculation is needed to sink $a =$ 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 $a =$.

Notations here (explicit loads and stores in grey then in black)

Please put the SSA-FRG in this first figure

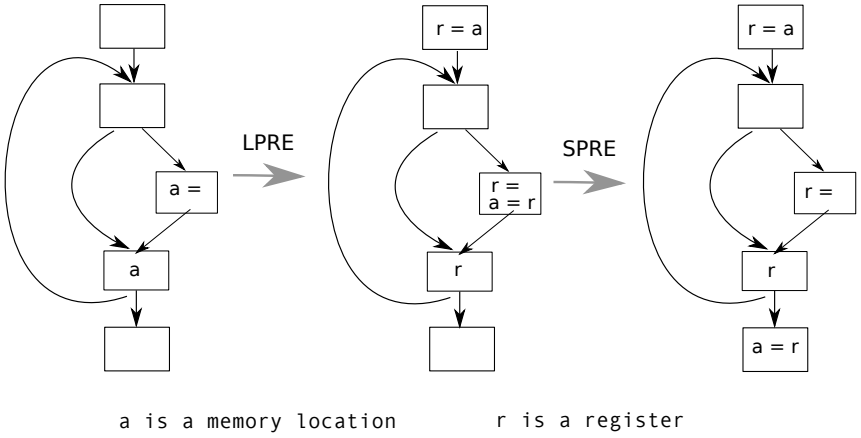


Fig. 1.11 Register promotion via load PRE followed by store PRE

1.5.3 Store Placement Optimization

As mentioned earlier, SPRE is the dual of LPRE. In the presence of store redundancies, the earlier occurrences are redundant. Code motion in SPRE will have the effect of moving stores forward with respect to the control flow graph. Any presence of (aliased) loads have 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, and we call this the *static single use* form. In SSU, use-def edges are factored at divergence points in the control flow graph. We call this factoring operator Λ. Each use of a variable establishes a new version (we say the load *uses* the version), and every store reaches exactly one load. The Λ is regarded as a use of a new version of the variable. The use post-dominates all the stores of its version. The left-hand-side of the Λ is the multiple definitions, each of which is post-dominated by their single uses. Add a reference to the SSI chapter here

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

It would be preferable to have both the code and the SSU-FRG just as you did for PRE. Somehow, you merge (a) & (b).

do you mind if we use the notation: (a2,a3)=SIGMA(a1)?

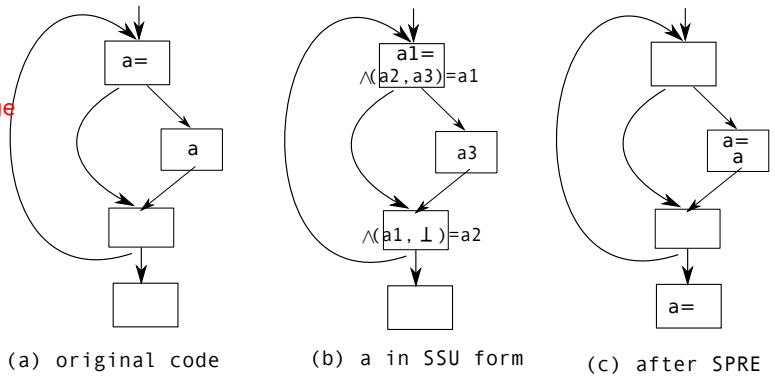


Fig. 1.12 Example of program in SSU form and the result of applying SSUPRE

I think it would be helpful to recall the insertion rules here (that explain why you insert the store for the a3 in the first sigma)

Figure 1.12 gives an example program with (b) being the SSU representation for the program in (a). (c) shows the result of applying SSUPRE to the code. The store can be sunk to outside the loop only if it is also inserted in the branch inside the loop that contains the load. The optimized code no longer exhibits any store redundancy.

Do you consider it to be an opened problem?

A limitation of the above SSUPRE algorithm is that it does not apply to indirect stores. It may be possible to extend the algorithm to make it work on indirect stores as well.

1.6 Redundancy via the Semantic Approach

Yes! See my remark above To be put before along with an example.

The redundant computations stored into a temporary introduced by PRE may be of different values because the same lexically identified expression may yield different results at different points in the program. Since PRE applies only to lexically identified expressions, it is not capable of recognizing redundant computations among lexically different expressions based on their semantics. On the other hand, under the semantic approach, redundant computations can be recognized among computations that are semantically determined to yield the same values.

1.6.1 Value Numbering

Computations are determined to yield the same value using *value numbering* analysis techniques. The term *value number* originated from the hash-based method developed by Cocke and Schwartz for recognizing when two expressions evaluate to the same value within a basic block [5]. 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 bottom up starting with the leaf nodes. Each internal node is hashed based on its operator and the value numbers of its operands. Two expressions with the same value number *must* evaluate to the same value at run-time.

Please give (inline) an example for expression (a+2)+b

I do not agree, different variable's version might (hopefully) have the same value number

When the program has been put into SSA form, value number can be extended to the global scope by assigning a unique value number to each variable version by virtue of the single definition property [19]. Additional refinements to hash-based global value numbering algorithms have been proposed by Briggs *et al* [2].

=> last section

A second approach for determining whether two expressions compute the same value that uses the partitioning method instead of hashing. First developed by Alpern *et al.* [1], the algorithe partitions all the expressions in the program into congruence classes. Values in the same congruence class are considered as evaluating to identical values. The algorithm is optimistic because when it starts, it puts all expressions based on the same operator into 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 used as 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 subdivision can occur. At this point, an enumeration of the resulting congruence classes can be used as value numbers. The detailed algorithm is shown in Figure 1.13. In the last **for** loop, $\phi \subset (s \cap \mathit{touched}) \subset s$ is the criterion for subdividing class s because the other members of s that do not have x at operand position p potentially compute to a different value. After the partition into n and the new s, if s is not in the worklist (i.e., processed already), the partitioning was already stable with respect to the old s, and we can add either n or the new s to

```
Place all values computed by the same opcode in the same congruence class
worklist ← set of all congruence classes
while worklist ≠ φ
    Select and delete an arbitrary congruence class c from worklist
    for each operand position p of a use of x ∈ c
        touched ← φ
        for each x ∈ c                                  use "y" to avoid confusion
            Add any expression in the program that uses x in position p to touched
        for each class s such that φ ⊂ (s ∩ touched) ⊂ s
            Create a new class n ← s ∩ tourhed
            s ← s − n
            if s ∈ worklist
                Add n to worklist
            else
                Add smaller of n and s to worklist
```

Fig. 1.13 The partitioning algorithm

the worklist to re-stabilize with respect to that split of s. Choosing the smaller one results in less overhead.

last section <=

Briggs *et al.* have also proposed additional refinements to the partitioning technique [2]. Partition-based algorithms do not supercede, but instead complement the hash-based algorithms. The most powerful value numbering algorithms are in fact based on a combination of the two approaches. Can you tell a bit more how you combine it? Is is just an intersection or something sophisticated. Any reference (for the last section of course :-))?

1.6.2 Redundancy Elimination via Value Numbering

So far, we have only talked about finding computations that compute the same value, but have not addressed how to use the results of such analysis to optimize the program. Two computations that compute the same value do not exhibit redundancy if they are not situated on the same execution path. Thus, it is logical to consider performing PRE separately for each value number.

If we consider the temporary t that will be introduced to store the redundant computations under value-number-based PRE, we can see that its value will stay the same throughout its lifetime. If there are ϕ's introduced for the temporary, they will be merging identical values, and we know from experience that such ϕ's are rare. A subset of such ϕ's is expected to come from PRE's insertions, and that implies that insertions introduced by value-number-based PRE are also rare.

Value-number-based PRE also has to deal with the additional 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 an 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, we focus only on eliminating full redundancies among computations that have the same value number[5].

Since full redundancy elimination (FRE) is a subset of PRE, we can adapt the SSAPRE algorithm to work on value-number-identified expressions and remove full redundancies among them. We call this adapted algorithm VNFRE.

In VNFRE, the Φ-Insertion step only needs to consider the iterated dominance frontiers. It is not necessary to consider the variable operands, because as long as the value number is the same, we do not have to consider when any variable operand's value is changed. The Renaming step is also simpler because we also ignore the variable operands. In fact, the only situations where new h-versions are introduced are at program entry (when the renaming stack is empty) and when encountering Φ's. With the FRG constructed, we perform the full availability analysis on the Φ's to identify the expressions that can be deleted.

In practice, with copy propagation, constant folding and SSAPRE having been performed earlier, there are not much optimization opportunities left for VNFRE to cater to. One useful application of VNFRE is in induction variable collescing. Code like the following, with multiple induction variables, could be the result of earlier optimizations like strength reduction:

```
i = 0;
j = 0;
while <cond> {
  i = i + 4;
  j = j + 4;
}
```

Value numbering will determine that at different points in the code, `i` and `j` always have the same value number. Performing VNFRE will yield the net effect of getting rid of the extra induction variables.

1.7 ~~Conclusion~~ Further readings / opened problems / etc

The close relationsihp between PRE and SSA arises because partial redundancies can be exposed by factoring at control flow merge points. The SSAPRE algorithm capitalizes on prior techniques developed for computing and manipulating SSA form. The SSAPRE framework also shows that the concept and techniques of SSA can be made to apply to any program constructs, not just variables. The constructed SSA graphs can then be used to efficiently perform sparse data flow propagation.

[5] This assumes PRE for lexically identified expressions have been applied, which would have removed the redundancies among lexically identified expressions that also have the same value number.

There are additional optimizations that can be implemented using the SSAPRE framework that we have not covered. They include code hoisting, register shrink-wrapping [3] and live range shrinking. Moreover, PRE has traditionally provided the context for integrating additional optimizations into its framework. They include operator strength reduction [12] and linear function test replacement [10]. As a result, PRE has become the most powerful and encompassing optimization framework in modern optimizing compilers.

References

1. B. Alpern, M. Wegman, and F. Zadeck. Detecting equality of values in programs. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–11, January 1988.
2. P. Briggs, K. Cooper, and L Simpson. Value numbering. *Software Practice and Experience*, 27(6):701–724, 1997.
3. 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, 1988.
4. F. Chow, S. Chan, R. Kennedy, S. Liu, R. Lo, and P. Tu. A new algorithm for partial redundancy elimination based on ssa form. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 273–286, 1997.
5. J. Cocke and J. Schwartz. Programming languages and their compilers. Technical report, Courant Institute of Mathematical Sciences, New York University, April 1970.
6. D. M. Dhamdhere. E-path_pre: partial redundancy made easy. *SIGPLAN Notices*, 37(8):53–65, 2002.
7. K. Drechsler and M. Stadel. A variation of knoop, rüthing and steffen's lazy code motion. *SIGPLAN Notices*, 28(5):29–38, 1993.
8. K. Kennedy. Safety of code motion. *International Journal of Computer Mathematics*, 3(2 and 3):117–130, 1972.
9. R. Kennedy, S. Chan, S. Liu, R. Lo, P. Tu, and F. Chow. Partial redundancy elimination in ssa form. *ACM Trans. Program. Lang. Syst.*, 21(3):627–676, 1999.
10. R. Kennedy, F. Chow, P. Dahl, S. Liu, R. Lo, P. Tu, and M. Streich. Strength reduction via ssapre. In *Proceedings of the Seventh International Conference on Compiler Construction*, 1988.
11. 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, 1992.
12. J. Knoop, O. Rüthing, and B. Steffen. Lazy strength reduction. *Journal of Programming Languages*, 1(1):71–91, 1993.
13. J. Knoop, O. Rüthing, and B. Steffen. Optimal code motion: theory and practice. *ACM Trans. Program. Lang. Syst.*, 16(4):1117–1155, 1994.
14. R. Lo, F. Chow, R. Kennedy, S. Liu, and P. Tu. Register promotion by sparse partial redundancy elimination of loads and stores. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 26–37, 1998.
15. V. Menon, N. Glew, B. Murphy, A. McCreight, T. Shpeisman, A. Adl-Tabatabai, and L. Petersen. A verifiable ssa program representation for aggressive compiler optimization. In *Proceedings of the 2006 POPL Conference*, pages 397–408, 2006.
16. E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, 1979.

17. B. Murphy, V. Menon, F. Schneider, T. Shpeisman, and A. Adl-Tabatabai. Fault-safe code motion for type-safe languages. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 144–154, 2008.
18. V.K. Paleri, Y.N. Srikant, and P. Shankar. Partial redundancy elimination: a simple, pragmatic and provably correct algorithm. *Science of Programming Programming*, 48(1):1–20, 2003.
19. 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, 1988.
20. J. Xue and Q. Cai. A lifetime optimal algorithm for speculative pre. *ACM Transactions on Architecture and Code Optimization*, 3(2):115–155, 2006.
21. J. Xue and J. Knoop. A fresh look at pre as a maximum flow problem, 2006.
22. H. Zhou, W.G. Chen, and F. Chow. An ssa-based algorithm for optimal speculative code motion under an execution profile. In *Proceedings of the ACM SIGPLAN '11 Conference on Programming Language Design and Implementation*, 2011.