Register Promotion by Sparse Partial Redundancy Elimination of Loads and Stores

Raymond Lo Fred Chow Robert Kennedy Shin-Ming Liu Peng Tu[1]

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

Abstract

An algorithm for register promotion is presented based on the observation that the circumstances for promoting a memory location's value to register coincide with situations where the program exhibits partial redundancy between accesses to the memory location. The recent SSAPRE algorithm for eliminating partial redundancy using a sparse SSA representation forms the foundation for the present algorithm to eliminate redundancy among memory accesses, enabling us to achieve both computational and live range optimality in our register promotion results. We discuss how to effect speculative code motion in the SSAPRE framework. We present two different algorithms for performing speculative code motion: the *conservative* speculation algorithm used in the absence of profile data, and the the *profile-driven* speculation algorithm used when profile data are available. We define the static single use (SSU) form and develop the dual of the SSAPRE algorithm, called SSUPRE, to perform the partial redundancy elimination of stores. We provide measurement data on the SPECint95 benchmark suite to demonstrate the effectiveness of our register promotion approach in removing loads and stores. We also study the relative performance of the different speculative code motion strategies when applied to scalar loads and stores.

1 Introduction

Register allocation is among the most important functions performed by an optimizing compiler. Prior to register allocation, it is necessary to identify the data items in the program that are candidates for register allocation. To represent register allocation candidates, compilers commonly use an unlimited number of *pseudo-registers* [CAC+81, TWL+91]. 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. Thus, using pseudo-registers simplifies the register allocator's job.

[1]Present address: Intel Corporation, 2200 Mission College Blvd., Santa Clara, CA 95052.

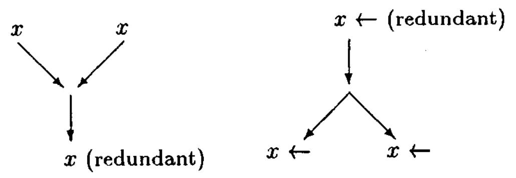
Optimization phases generate pseudo-registers to hold the values of computations that can be reused later, like common subexpressions and loop-invariant expressions. Variables declared with the *register* attribute in the C programming language, together with local variables determined by the compiler to have no alias, can be directly represented as pseudo-registers. All remaining register allocation candidates have to be assigned pseudo-registers through the process of *register promotion* [CL97]. Register promotion identifies sections of code in which it is safe to place the value of a data object in a pseudo-register. Register promotion is regarded as an optimization because instructions generated to access a data object in register are more efficient than if it is not in a register. If later register allocation cannot find a real register to map to a pseudo-register, it can either spill the pseudo-register to memory or re-materialize it, depending on the nature of the data object.[2]

This paper addresses the problem of register promotion within a procedure. We assume that earlier alias analysis has already identified the points of aliasing in the program, and that these aliases are accurately characterized in the static single assignment (SSA) representation of the program [CCL+96]. The register promotion phase inserts efficient code that sets up data objects in pseudo-registers, and rewrites the program code to operate on them. The pseudo-registers introduced by register promotion are maintained in valid SSA form. Our targets for register promotion include scalar variables, indirectly accessed memory locations and program constants. Since program constants can only be referenced but not stored into, they represent only a subset of the larger problem of register promotion for memory-resident objects. For convenience, we choose to exclude them from discussion for the rest of this paper, even though our solution does apply to them.[3]

Register promotion is relevant only to objects that need to be memory-resident in some part of the program. Global variables are targets for register promotion because they reside in memory between procedures. Aliased variables need to reside in memory at the points of aliasing. Before register promotion, we can regard the register promotion candidates as being memory-resident throughout the program. As a re-

[2]For a program variable that has an allocated home location, spilling its pseudo-register to the home location produces more efficient code than spilling to a new temporary location. Spilling-to-home and re-materialization can be regarded as the reverse of register promotion [Bri92].

[3]In applying register promotion to program constants, the process of materializing a program constant in a register corresponds to the loading of a variable from memory to a register.



(a) loads of x (b) stores to x

Figure 1: Duality between load and store redundancies

sult, there is a load operation associated with each of their uses, and there is a store operation associated with each assignment to them. Hence register promotion can be modeled as two separate problems: partial redundancy elimination of loads, followed by partial redundancy elimination of stores.

Partial redundancy elimination (PRE) is a powerful optimization concept first developed by Morel and Renvoise [MR79]. By performing data flow analysis on a computation, it determines where in the program to insert the computation. These insertions in turn cause partially redundant computations to become fully redundant, and therefore safe to delete. Knoop *et al.* came up with a different PRE algorithm called lazy code motion that improves on Morel and Renvoise's results [KRS92, DS93, KRS94a]. The result of lazy code motion is optimal: the number of computations cannot be further reduced by safe code motion, and the lifetimes of the pseudo-registers introduced are minimized.

Our team at Silicon Graphics has recently developed a new algorithm to perform PRE based on SSA form, called SSAPRE [CCK+97]. SSAPRE achieves the same optimal result as lazy code motion. Applying SSAPRE to loads thus has the effects of moving them backwards with respect to the control flow while inserting them as late as possible. The development of SSAPRE was motivated by the fact that traditional data flow analysis based on bit vectors does not interface well with the SSA form of program representation. In contrast, the SSAPRE algorithm takes advantage of the SSA representation and intrinsically produces its optimized output in SSA form. It does not use bit vectors and instead works on one candidate at a time, using the built-in use-def edges in SSA to propagate data flow information. The SSAPRE algorithm exhibits the same attributes of sparseness inherent in other SSA-based optimization algorithms. The entire program is maintained in valid SSA form as SSAPRE iterates through the PRE candidates.

For the sake of recognizing redundancy, loads behave like ordinary 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(a) and (b). The PRE of stores problem, also called ***partial dead store elimination***, can thus be treated as the dual of the PRE of loads problem. Performing PRE of stores thus has the effects of moving stores forward while inserting them as early as possible. By combining the effects of the PRE of loads and stores, our register promotion approach results in optimal placements of loads and stores while minimizing the live ranges of the pseudo-registers.

The rest of this paper is organized as follows. Section 2 surveys previous works related to register promotion and partial dead store elimination. Section 3 gives an overall perspective of our register promotion approach. Section 4 discusses our algorithm for the PRE of loads. Section 5 discusses how speculative code motion can be incorporated into the SSAPRE framework, and presents two different strategies for performing speculation, depending on whether profile data are available. In Section 6, we develop and present our algorithm for the PRE of stores. In Section 7, we provide measurement data to demonstrate the effectiveness of the techniques presented in removing loads and stores, and to study the relative performance of the different speculative code motion strategies when applied to scalar loads and stores. We conclude in Section 8.

2 Related Work

Different approaches have been used in the past to perform register promotion. Chow and Hennessy use data flow analysis to identify the live ranges where a register allocation candidate can be safely promoted [CH90]. Because their global register allocation is performed relatively early, at the end of global optimization, they do not require a separate register promotion phase. Instead, their register promotion is integrated into the global register allocator, and profitable placement of loads and stores is performed only if a candidate is assigned to a real register. In optimizing the placement of loads and stores, they use a simplified and symbolic version of PRE that makes use of the fact that the blocks that make up each live range must be contiguous.

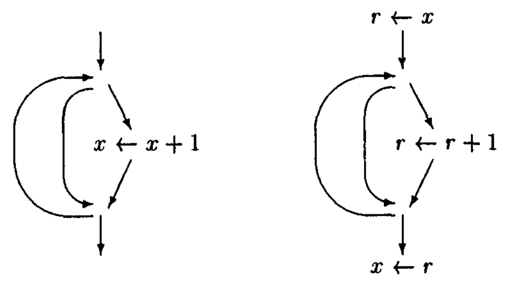
Cooper and Lu use an approach that is entirely loop-based [CL97]. By scanning the contents of the blocks comprising each loop, they identify candidates that can be safely promoted to register in the full extent of the loop. The load to a pseudo-register is generated at the entry to the outermost loop where the candidate is promotable. The store, if needed, is generated at the exit of the same loop. Their algorithm handles both scalar variables and pointer-based memory accesses where the base is loop-invariant. Their approach is all-or-nothing, in the sense that if only one part of a loop contains an aliased reference, the candidate will not be promoted for the entire loop. They do not handle straight-line code, relying instead on the PRE phase to achieve the effects of promotion outside loops, but it is not clear if their PRE phase can handle stores appropriately.

Dhamdhere was first to recognize that register promotion can be modeled as a problem of code placement for loads and stores, thereby benefiting from the established results of PRE [Dha88, Dha90]. His Load-Store Insertion Algorithm (LSIA) is an adaptation of Morel and Renvoise's PRE algorithm for load and store placement optimization. LSIA solves for the placements of both loads and stores at the same time.

The PRE of stores in the context of register promotion can be viewed as another approach to partial dead store elimination (PDE), for which numerous algorithms have been described. Chow applied the dual of Morel and Renvoise's PRE algorithm to the optimization of store statements [Cho83]. After solution of the data flow equations in bit vector form, an insertion pass identifies the latest insertion point for each store statement taking into account any possible modification of the right hand side expression. The algorithm by Knoop *et al.* is also PRE-based, but they separate it into an elimination step and a sinking step, and iterate them exhaustively so as to cover second order effects [KRS94b]. Their algorithm is thus more expensive than straight PRE. To additionally cover *faint* code elimination,[4] they use *slotwise* solution of the data flow equations [DRZ92].

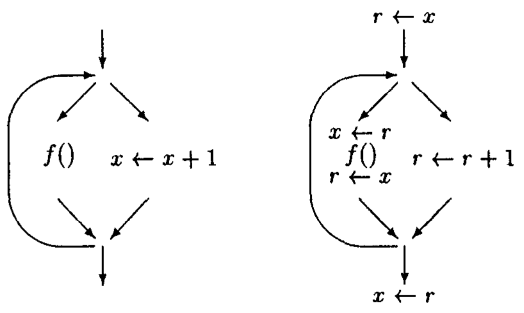
The PRE-based approaches to PDE do not modify the control flow structure of the program, but this limits the partial dead stores that can be removed. Non-PRE-based

[4]A store is faint if it is dead or becomes dead after some other dead stores have been deleted.



(a) original code (b) speculative load/store inserted

Figure 2: Speculative insertion of load and store



(a) original code (b) x promoted to register

Figure 3: Load and store inserted on unpromoted path

PDE algorithms can remove additional partial dead stores by modifying the control flow. In the revival transformation [FKCX94], a partially dead statement is detached from its original place in the program and reattached at a later point at which it is minimally dead. In cases where movement of a single store is not possible, the transformation will move a superstructure that includes other statements and branches. However, the coverage of the revival transformation is limited because it cannot be applied across loop boundaries. The algorithm as presented also does not consider situations that require multiple reattachment points to remove a partial dead store.

A PDE approach using slicing transformations was recently proposed by Bodík and Gupta [BG97]. Instead of moving partially dead statements, they take the approach of predicating them. The predication embeds the partially dead statement in a control flow structure, determined through program slicing, such that the statement is executed only if the result of the statement is eventually used. A separate branch deletion phase restructures and simplifies the flow graph. Their algorithm works on one partially dead statement at a time. Since the size of the code may grow after the PDE of each statement, complete PDE may take exponential time, and results in massive code restructuring. The vastly different code shape can cause additional variation in program performance.

Another PDE algorithm described by Gupta *et al.* [GBF97a] uses predication to enable code sinking in removing partial dead stores. The technique uses path profiling information to target only statements in frequently executed paths. A cost-benefit data flow analysis technique determines the profitability of sinking, taking into account the frequencies of each path considered. The same approach is used in [GBF97b] to speculatively hoist computations in PRE. Decisions to speculate are made locally at individual merge or split points based on the affected paths. Acyclic and cyclic code are treated by different versions of the algorithm.

3 Overview of Approach

In our PRE-based approach to register promotion, we apply the PRE of loads first, followed by the PRE of stores. This is different from Dhamdhere's LSIA, which solves for the placements of both loads and stores at the same time. Our 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. Decoupling the treatments of loads and stores also allows us to use an algorithm essentially unchanged from the base PRE algorithm. Our approach is SSA-based and worklist-driven. We cannot benefit from the parallelism inherent in bit vector operations, but we make up for that by doing data flow analysis on the sparse SSA representation, which takes fewer steps. Handling one candidate at a time allows easier, more intuitive and flexible implementation. When there are fewer candidates to work on, our approach will finish earlier, whereas a bit-vector-based approach always requires some material fixed cost. Our approach is thus more cost-effective, because the number of candidates for register promotion in a procedure often shows wide variation.

An advantage of SSAPRE related to the optimization of loads is that, given our SSA program representation that encodes alias information using virtual variables [CCL+96], it is easy to perform additional context-sensitive alias analyses during SSAPRE's *Rename* step to expose more redundancy among loads that have potential aliases. In situations where there is a chain of aliasing stores, our sparse approach can stop after identifying the first aliasing store. In contrast, traditional bit-vector-based approaches would have to analyze the sequence completely in order to initialize the bit vectors for data flow analyses. Hence, in programs with many aliased loads and stores, SSAPRE is often faster than traditional bit-vector-based PRE.

A further advantage of using the SSAPRE framework is that, given an existing implementation of SSAPRE for general expressions, only a small effort is needed to obtain coverage for the PRE of indirect loads and scalar loads. In our case, most of the additional implementation effort was spent in implementing the PRE of stores.

There is one important difference between PRE-based and non-PRE-based register promotion approaches. PRE by its nature does not perform speculative code motion, but in the area of register promotion, it is sometimes beneficial to insert loads and stores speculatively. In Figure 2(a), it is highly desirable to promote variable x to register for the duration of the loop, but the branch in the loop that contains the accesses to x may not be executed, and promoting x to register, as shown in Figure 2(b), is speculative. In this respect, non-PRE-based promotion approaches have an advantage over PRE-based approaches. On the other hand, it is not always good to insert loads and stores speculatively. In the slightly different example of Figure 3(a), the call $f()$ contains aliases to variable x. Promoting x to register requires storing it before the call and reloading it after the call (Figure 3(b)). If the path containing the call is executed more frequently than the path containing the increment to x, promoting x to register will degrade the performance of the loop. We have developed new techniques to control speculation in the SSAPRE framework.

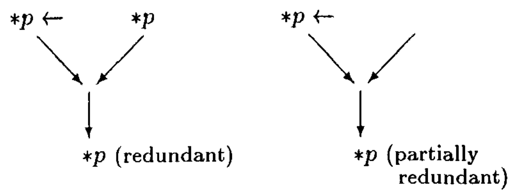


Figure 4: Redundant loads after stores

To perform the PRE of stores, we develop the dual of the SSAPRE algorithm called SSUPRE. Because we treat PDE in the context of register promotion, we do not take into account the right hand side of the store. We view each store statement $x \leftarrow \langle\text{expr}\rangle$ as if it is made up of the sequence:

$$r \leftarrow \langle\text{expr}\rangle$$
$$x \leftarrow r$$

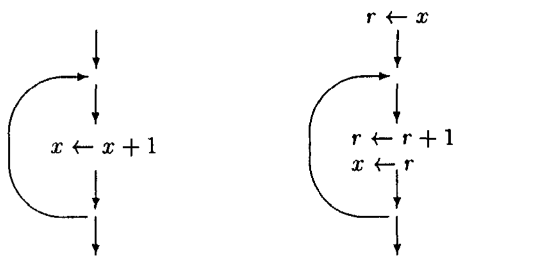
PDE is then applied purely to the store $x \leftarrow r$. This allows greater movement of the store, because it is not blocked by any modification to $\langle\text{expr}\rangle$, while simplifying our algorithm. We also do not need to be concerned with second order effects, because doing the earlier PRE of loads effectively removes most of the loads that can block the movement of stores. Before we perform register promotion, we invoke the standard SSA-based dead store elimination algorithm [CFR+91], which deletes all dead or faint stores. We cannot eliminate stores that become partially dead after some other partially dead stores have been removed; if desired, they can still be eliminated by iterating the PRE of stores phase.

4 Load Placement Optimization

Before register promotion, there is a load associated with each reference to the variable. Applying PRE to loads removes redundancy among the loads and introduces pseudo-registers to hold the values of redundant loads to be reused. The same holds for indirect load operations in the program. The SSAPRE algorithm can be applied to both types of loads without much modification. We now give a brief overview of the SSAPRE algorithm. The reader is referred to [CCK+97] for a full discussion of SSAPRE.

SSAPRE performs PRE on one program computation at a time. For a given program computation, E, SSAPRE consists of six separate steps. The first two steps, (1) Φ-*Insertion* and (2) *Rename*, construct the SSA form for the hypothetical temporary h that represents the value of E. The next two steps, (3) *DownSafety* and (4) *WillBeAvail*, perform sparse computation of global data flow attributes based on the SSA graph for h. The fifth step, (5) *Finalize*, determines points in the program to insert computations of E, marks computations that need to be saved and computations that are redundant, and determines the use-def relationship among SSA versions of the real temporary t that will hold the value of E. The last step, (6) *CodeMotion*, transforms the code to form the optimized output.

In our SSA representation [CCL+96], indirect loads are in the form of expression trees, while direct loads are leaves in the expression trees. SSAPRE processes the operations in an expression tree bottom-up. If two occurrences of an indirect load, $*(a_1 + b_1)$ and $*(a_2 + b_2)$, have partial redundancy between them, the two address expressions $(a_1 + b_1)$ and $(a_2 + b_2)$ must also have partial redundancy between them. Because of the bottom-up processing order, by the time SSAPRE works on the indirect loads, the address expressions must have been converted to temporaries t_1 and t_2.



(a) original code (b) after PRE of loads

Figure 5: Load placement via store-load interaction

Hence, SSAPRE only needs to handle indirect loads whose bases are (or have been converted to) leaves.

A store of the form $x \leftarrow \langle\text{expr}\rangle$ can be regarded as being made up of the sequence:

$$r \leftarrow \langle\text{expr}\rangle$$
$$x \leftarrow r$$

Because the pseudo-register r contains the current value of x, any subsequent occurrences of the load x can reuse the value from r, and thus can be regarded as redundant. The same observations apply to indirect stores, replacing x by $*p$. Figure 4 gives examples of loads made redundant by stores.

The implication of this store-load interaction is that we have to take into account the occurrences of the stores when we perform the PRE of loads. During PRE on the loads of x, $x \leftarrow$ is called a *left* occurrence. The Φ-*Insertion* step will also insert Φ's at the iterated dominance frontiers of left occurrences. In the *Rename* step, a left 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.

In the *CodeMotion* step, if a left occurrence is marked *save*, the corresponding store statement will be split into two statements:

$$x \leftarrow \langle\text{expr}\rangle \Longrightarrow \begin{array}{l} t_1 \leftarrow \langle\text{expr}\rangle \\ x \leftarrow t_1 \end{array}$$

The placement of the new store $x \leftarrow t_1$ will be optimized by the PRE of stores performed after the PRE of loads.

The importance of store-load interaction is illustrated by Figure 5. Ordinarily, we cannot move the load of x out of the loop because x's value is changed inside the loop. Recognizing $x \leftarrow$ as a left occurrence exposes partial redundancy in the load of x. PRE in turn moves the load of x to the loop header. The store to x will be moved to the loop exit when we perform PRE of stores later (see Section 6).

In performing SSAPRE for direct loads, the Φ-*Insertion* and *Rename* steps can be streamlined by taking advantage of the variable being already in SSA form. We can just map the ϕ's and SSA versions of the variable to the Φ's and h-versions of its load operation.

5 Speculative Code Motion

In its basic framework, PRE does not allow the insertion of any computation at a point in the program where the computation is not down-safe (*i.e.*, anticipated). This is necessary to ensure the safety of the code placement. Speculation corresponds to inserting computations during SSAPRE at Φ's where the computation is not down-safe. We can

```
function Has_⊥_Φ_opnd(X, F)
  if (X is ⊥)
    return true
  if (X not defined by Φ)
    return false
  if (X = F)
    return false
  if (visited(X))
    return false
  visited(X) ← true
  for each operand opnd of X do
    if (Has_⊥_Φ_opnd(opnd, F))
      return true
  return false
end Has_⊥_Φ_opnd

function Can_speculate(F)
  for each Φ X in the loop do
    visited(X) ← false
  for each back-edge operand opnd of F do
    if (Has_⊥_Φ_opnd(opnd, F))
      return false
  return true
end Can_speculate
```

Figure 6: Algorithm for *Can_speculate*

accomplish this effect by selectively marking non-down-safe Φ's as *down_safe* in the *DownSafety* step of SSAPRE. In the extreme case, we can mark *all* Φ's as *down_safe* in the *DownSafety* step of SSAPRE. We refer to the resulting code motion as *full speculation*.

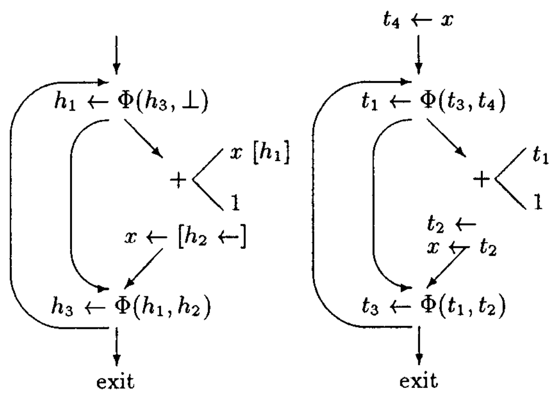
A necessary condition for speculative code motion in general is that the operation moved must not cause any unmaskable exception. Direct loads can usually be speculated. However, indirect loads from unknown pointer values need to be excluded, unless the hardware can tolerate them.

Speculation may or may not be beneficial to program performance, depending on which execution paths are taken more frequently. Thus, it is best to base speculation decisions on the profile data of the program being compiled. In the absence of profile data, there are situations where it is often desirable to speculatively insert loads and stores, as we have discussed with respect to Figure 2. In this section, we present two different speculative code motion strategies depending on whether profile data are available.

5.1 Conservative Speculation

The conservative speculation strategy is used when profile data are not available. Under this situation, we restrict speculative code motion to moving loop-invariant computations out of single-entry loops.

We base our analysis on the Φ located at the start of the loop body. We perform speculative insertion at the loop header only if no other insertion inside the loop is needed to make the computation fully available at the start of the loop body. The algorithm for this analysis is given by function *Can_speculate* shown in Figure 6. For the Φ F, we identify the Φ operands that correspond to the back edges of the loop, and call function *Has_⊥_Φ_opnd* for each of these Φ operands. Since ⊥ represents potential insertion points, *Has_⊥_Φ_opnd* returning false will indicate that the computation is available at that back edge without requiring any insertion other than the one at the operand of F that corresponds to the loop entry. If the checks succeed, we mark F as *down_safe*. Figure 7 shows the results of applying this algorithm to the program of Figure 2(a). The



$h_1 \leftarrow \Phi(h_3, \perp)$
$+$
$x\ [h_1]$
1
$x \leftarrow [h_2 \leftarrow]$
$h_3 \leftarrow \Phi(h_1, h_2)$
exit
$t_4 \leftarrow x$
$t_1 \leftarrow \Phi(t_3, t_4)$
$+$
t_1
1
$t_2 \leftarrow$
$x \leftarrow t_2$
$t_3 \leftarrow \Phi(t_1, t_2)$
exit

(a) SSA graph for h (b) resulting program

Φ for h_1 marked as *down-safe*

Figure 7: Speculative load placement example

algorithm marks the Φ corresponding to h_1 in Figure 7(a) as *down_safe*. Figure 7(b) shows the program after the load has been speculatively moved out of the loop. Subsequent application of speculative code motion in the PRE of stores will move the store out of the loop to yield the result shown in Figure 2(b).

5.2 Profile-driven Speculation

Without knowledge of execution frequency, any speculation can potentially hurt performance. But when execution profile data are provided, it is possible to tailor the use of speculation to maximize run-time performance for executions of the program that match the given profile. The optimum code placement lies somewhere between no speculation and full speculation. Code placement with no speculation corresponds to the results obtained by traditional PRE. Code placement with full speculation corresponds to the results of SSAPRE if *all* Φ's are marked *down_safe* in the *DownSafety* step. The problem of determining an optimum placement of loads and stores can be expressed as instances of the integer programming problem, but we know of no practical algorithm for solving it. Instead of aiming for the optimal solution, we settle on a practical, versatile and easy-to-implement solution that never performs worse than no speculation, subject to accuracy of the profile data.

In our approach, the granularity for deciding whether to speculate is each connected component of the SSA graph formed by SSAPRE. For each connected component, we either perform full speculation, or do not speculate at all. Though this limits the possible placements that we can consider, it enables us to avoid the complexity of finding and evaluating the remaining placement possibilities. When the connected components are small, we usually get better results, since we miss fewer placement possibilities. The connected components for expressions are generally quite small.

Our profile-driven speculative code motion algorithm works by comparing the performance with and without speculation using the basic block frequency data provided by the execution profile. The overall technique is an extension of the SSAPRE algorithm. Pseudo-code is given in Figure 8. Procedure *SSAPRE_with_profile* gives the overall phase structure for profile-driven SSAPRE. We start by performing regular SSAPRE with no speculation. Before the *CodeMotion* step, we call function *Speculating()*, which determines whether there is any connected component in the

```
procedure Compute_speculation_savings
  for each connected component C in SSA graph
    savings[C] ← 0
  for each real occurrence R in SSA graph
    if (Reload(R) = false and R is defined by a Φ F)
      savings[Connected_component(F)] += freq(R)
  for each Φ occurrence F in SSA graph
    for each operand opnd of F
      if (Insert(opnd) = true) {
        if (opnd is defined by a Φ)
          savings[Connected_component(F)] += freq(R)
      }
      else if (opnd is ⊥)
        savings[Connected_component(F)] -= freq(opnd)
end Compute_speculation_savings

function Speculating()
  has_speculation ← false
  remove Φ's that are not partially available or
                          partially anticipated
  identify connected components in SSA graph
  Compute_speculation_savings()
  for each connected component C in SSA graph
    if (savings[C] > 0) {
      mark all Φ's in C down_safe
      has_speculation ← true
    }
  return has_speculation
end Speculating

procedure SSAPRE_with_profile
  Φ-Insertion Step
  Rename Step
  DownSafety Step
  WillBeAvail Step
  Finalize Step
  if (Speculating()) {
    WillBeAvail Step
    Finalize Step
  }
  CodeMotion Step
end SSAPRE_with_profile
```

Figure 8: Profile-driven SSAPRE Algorithm

computation being processed that warrants full speculation. If *Speculating()* returns true, it will have marked the relevant Φ's as *down_safe*. Thus, it is necessary to re-apply the *WillBeAvail* and *Finalize* steps, which yield the new code placement result with speculation. The last step, *CodeMotion*, works the same way regardless of whether speculation is performed or not.

Speculating() is responsible for determining if full speculation should be applied to each connected component in the SSA graph. First it prunes the SSA graph by removing Φ's where the computation is not partially available or not partially anticipated. Φ's where the computation is not partially available are never best insertion points because some later insertion points yield the same redundancy and are better from the point of view of the temporary's live range. Insertions made at Φ's where the computation is not partially anticipated are always useless because they do not make possible any deletion. After removing these Φ's and deleting all references to them, *Speculating()* partitions the SSA graph into its connected components. Next, procedure *Compute_speculation_savings* determines whether speculation can reduce the dynamic counts of the computation on a per-connected-component basis, using the results of SSAPRE with no speculation as the baseline.

Given a connected component of an SSA graph where the computation is partially available throughout, it is straightforward to predict the code placement that corresponds to full speculation. Since we regard all Φ's in the component as down-safe, the *WillBeAvail* step will find that *can_be_avail* holds for all of them. The purpose of the computation of *Later* in the *WillBeAvail* step is only for live range optimality, and does not affect computational optimality. If we ignore the *Later* property, the *Finalize* step will decide to insert at all the ⊥ operands of the Φ's. In addition, the insertions will render fully redundant any real occurrence within the connected component whose h-version is defined by Φ.

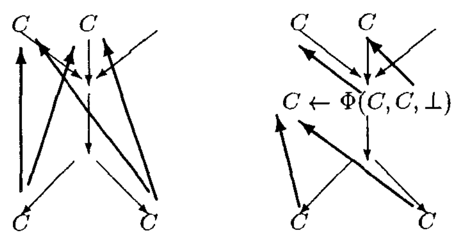
By predicting the code placement for full speculation, *Compute_speculation_savings* can compute the benefits of performing full speculation for individual connected components and store them in the array *savings*, indexed by the connected components. *savings* is the sum of the dynamic counts of the real occurrences deleted and any non-speculative SSAPRE insertions suppressed due to full speculation, minus the dynamic counts for the new insertions made by full speculation. Procedure *Compute_speculation_savings* iterates through all real and Φ occurrences in the SSA graph. Whenever the algorithm encounters a non-deleted real occurrence that is defined by a Φ, it increases *savings* for the connected component by the execution frequency of the real occurrence. Deletions made by SSAPRE are passed over because those deletions would have been done with or without the speculation part. Other real occurrences are not affected by full speculation. For each Φ occurrence, procedure *Compute_speculation_savings* iterates through its operands. If the Φ operand is marked *insert* and is defined by another Φ, the algorithm also increases *savings* for the connected component by the execution frequency of the Φ operand because full speculation will hoist such insertions to earlier Φ's. If the Φ operand is ⊥, the algorithm decreases *savings* for the connected component by the execution frequency for the Φ operand because under full speculation, insertions will be performed at these Φ operands.

After procedure *Compute_speculation_savings* returns, *Speculating()* iterates through the list of connected components. If the tallied result in *savings* is positive for a connected component, it means speculative insertion is profitable, and *Speculating()* will effect full speculation by marking all Φ's in the connected component as *down_safe*.

With our profile-driven speculation algorithm, *speculation is performed only in those connected components where it is beneficial*. In contrast to the technique by Gupta *et al.* [GBF97a], our decisions are made globally based on a per-connected-component basis in the SSA graph. In the example of Figure 3, our algorithm will promote x to a register if the execution count of the path containing the call is lower than that of the path containing the store. In the absence of profile data, our conservative speculation algorithm of Section 5.1 will not trigger the speculative code motion in this example because it requires insertion in the body of the loop.

6 Store Placement Optimization

In this section, we develop our algorithm to perform PRE of stores (or PDE). We first present the underlying principles behind our sparse approach to PRE. We then relate and contrast the characteristics between loads and stores to establish the duality between load redundancy and store redundancy. Given this duality, we then describe our SSUPRE



(a) before factoring (b) after factoring

⟵ redundancy edge

⟵ control flow edge

C is either an expression or a load

Figure 9: Factoring of redundancy edges

algorithm that performs PRE of stores.

6.1 Foundation of Sparse PRE

Suppose we are working on a computation C, which performs an operation to yield a value. Let us focus on the occurrence C_1 with respect to which other occurrences are redundant, and assume there is no modification of the value computed by C_1 in the program.[5] Any occurrence of C in the region of the control flow graph dominated by C_1 is fully redundant with respect to C_1; an occurrence of C outside this region may be partially redundant with respect to C_1. The earliest possible strictly partially redundant occurrences of C with respect to C_1 are in the dominance frontier of C_1. Dominance frontiers are also the places where ϕ operators are required in *minimal* SSA form [CFR+91], intuitively indicating that there are common threads between PRE and properties of SSA form.

Our sparse approach to PRE, as exemplified by SSAPRE, relies on a representation that can directly expose partial redundancy; such a representation can be derived as follows. Suppose an occurrence C_2 is partially redundant with respect to C_1. We represent this redundancy by a directed edge from C_2 to C_1.

In general, if the computation C occurs many times throughout the program, there will be many such edges. The relation represented by these edges is many-to-many, because an occurrence can be redundant with respect to multiple occurrences. We factor these redundancy edges by introducing a Φ operator at control flow merge points in the program. The effect of this factoring is to remove the many-to-many relationships, and convert them to many-to-one so that each occurrence can only be redundant with respect to a single occurrence, which may be a Φ occurrence. In the factored form, each edge represents full redundancy because the head of each edge must dominate the tail of the edge after the factoring. Strict partial redundancy is exposed whenever there is a missing incoming edge to a Φ, *i.e.*, a $\perp$ Φ operand (Figure 9).

Having identified this sparse graph representation that can expose partial redundancy, we need a method to build the representation. Because the representation shares many of the characteristics of SSA form, the method to build this sparse graph closely parallels the standard SSA construction algorithm.[6] The *Φ-Insertion* step inserts Φ's at the iterated dominance frontiers of each computation to serve as anchor points for placement analysis. In the *Rename* step, we assign SSA versions to occurrences according to the values they compute. The resulting SSA versions encode the redundancy edges of the sparse graph as follows: if an occurrence has the same SSA version as an earlier occurrence, it is redundant with respect to that earlier occurrence.

In our sparse approach to PRE, the next two steps, *DownSafety* and *WillBeAvail*, perform data flow analysis on the sparse graph. The results enable the next step, *Finalize*, to pinpoint the locations in the program to insert the computation. These insertions make partially redundant occurrences become fully redundant, which are marked. At this point, the form of the optimized output has been determined. The final step, *CodeMotion*, transforms the code to form the optimized program.

6.2 Duality between Loads and Stores

For register promotion, we assign a unique pseudo-register r for each memory location involved in load and store placement optimization. For indirect loads and stores, we assign a unique pseudo-register for each lexically identical address expression. The discussion in this section applies to both direct and indirect loads and stores, though we use direct loads and stores as examples in the discussion.

A load of the form $r \leftarrow x$ is fully (partially) redundant if the load is fully (partially) available. Thus, given two occurrences of the loads, the later occurrence is the redundant occurrence. On the other hand, a store, of the form $x \leftarrow r$, is fully (partially) redundant if the store is fully (partially) anticipated. Given two occurrences of the stores, the earlier occurrence is the redundant occurrence (Figure 1). As a result, redundancy edges for loads point backwards with respect to the control flow, while redundancy edges for stores point forward. The availability and anticipation of a load is killed when the memory location is modified. On the other hand, the availability and anticipation of a store is killed when the memory location is used; the movement of an available store is blocked additionally by an aliased store.

A load of x is fully redundant with respect to an earlier load of x only if the earlier load occurs at a place that dominates the current load, because this situation implies the earlier load must be executed before control flow reaches the current load. Since redundancy edges are factored across control flow merge points, the targets of the new edges always dominate their sources. All the edges now represent full load redundancies, and partial load redundancies are exposed when there are $\perp$ operands in the factoring operator Φ. A store to x is fully redundant with respect to a later store to x only if the later store occurs at a place that post-dominates the current store, because this implies the later store must eventually be executed after the current store. Since redundancy edges are factored across control flow split points, the targets of the new edges always post-dominates their sources. All the edges now represent full store redundancies, and partial store redundancies are exposed when there are $\perp$ operands in the factoring operator Λ. In performing PRE, we move loads backward with respect to the control flow and insert them as late as possible to minimize r's lifetime; for stores, however, we move them forward and insert them as early as possible to minimize r's lifetime.

We define *static single use* (SSU) form to be the dual of SSA form; in SSU form each use of a variable establishes a new version (we say the load *uses* the version), and every store reaches exactly one load. Just as the SSA factoring

[5] We use subscript here purely to identify individual occurrences; they are not SSA versions.

[6] SSA form is actually a kind of factored use-def representation; discussion of this can be found in [Wol96].

	load: $r \leftarrow x$	store: $x \leftarrow r$
when redundant?	available	anticipated
which redundant?	later occurrence	earlier occurrence
direction of edge	backward	forward
what kills?	store	load
what blocks motion?	store	load, aliased store
fully redundant if	dominated	post-dominated
where to factor?	merge points	split points
factoring operation	$h_3 \leftarrow \Phi(h_1, h_2)$	$\Lambda(h^2, h^3) \leftarrow h^1$
movement in PRE	backward	forward
minimize r's lifetime	late insertion	early insertion

Table 1: Duality between load and store redundancies

operator Φ is regarded as a definition of the corresponding variable and always defines a new version, the SSU factoring operator Λ is regarded as a use of its variable and always establishes (uses) a new version. Each use post-dominates all the stores of its version. Just as SSA form serves as the framework for the SSAPRE algorithm, SSU form serves as the framework for our algorithm for eliminating partial redundancy among stores, which we call SSUPRE. We annotate SSU versions using superscripts.

Table 1 summarizes our discussion on the duality between load and store redundancies.

6.3 SSUPRE Algorithm

Having established the duality between load and store redundancies, we are now ready to give the algorithm for our sparse approach to the PRE of stores. For a general store statement, of the form $x \leftarrow \langle\text{expr}\rangle$, we view it as if it is made up of the sequence:

$$\begin{aligned} &r \leftarrow \langle\text{expr}\rangle \\ &x \leftarrow r \end{aligned}$$

PRE is only applied to the store $x \leftarrow r$, where x is a direct or indirect store and r is a pseudo-register. For maximum effectiveness, the PRE of stores should be performed after the PRE of loads, because the PRE of loads will convert many loads into register references so they would not block the movement of the stores, as shown in Figure 2.

Our SSUPRE algorithm for the PRE of stores is transcribed and dualized from the SSAPRE algorithm, except that it cannot exploit the SSA representation of the input in the same way as SSAPRE. As a result, it is less efficient than SSAPRE on a program represented in SSA form. To achieve the same efficiency as SSAPRE, it would have been necessary for the input program to be represented in SSU form. Such a representation of the input is not practical because it would benefit only this particular optimization, so SSUPRE constructs the required parts of SSU form on demand.

Like SSAPRE, the SSUPRE algorithm is made up of six steps, and is applicable to both direct and indirect stores. It works by constructing the graph of factored redundancy edges of the stores being optimized, called the SSU graph. The first two steps, Λ-*Insertion* and *Rename*, work on all stores in the program at the same time while conducting a pass through the entire program. The remaining steps can be applied to each store placement optimization candidate one at a time.

6.3.1 The Λ-*Insertion* Step

The purpose of Λ is to expose the potential insertion points for the store being optimized. There are two different scenarios for Λ's to be placed. First, Λ's have to be placed at the iterated post-dominance frontiers of each store in the program. Second, Λ's also have to be placed when a killed store reaches a split point; since stores are killed by loads, this means Λ's have to be placed at the iterated post-dominance frontiers of each load (including aliased load) of the memory location. In Figure 10(b), the Λ at the bottom of the loop body is inserted due to its being a post-dominance frontier of $x \leftarrow$ inside the loop, and the Λ at the split in the loop body is inserted due to its being a post-dominance frontier of the use of x in one branch of the split. Λ insertion is performed in one pass through the entire program for all stores that are PRE candidates.

6.3.2 The *Rename* Step

This step assigns SSU versions to all the stores. Each use is assigned a new SSU version, which is applied to the stores that reach it. Each Λ is assigned a new SSU version because we regard each Λ as a use. The result of renaming is such that any control flow path that includes two different versions must cross an (aliased) use of the memory location or a Λ.

Renaming is performed by conducting a preorder traversal of the post-dominator tree, beginning at the exit points of the program. We maintain a renaming stack for every store that we are optimizing. When we come across an aliased load (use) or Λ, we generate a new SSU version and push it onto the stack. When we come to a store, we assign it the SSU version at the top of its stack and also push it onto the stack. Entries on the stacks are popped as we back up the blocks containing the uses that generate them. The operands of Λ's are renamed at the entries of their corresponding successor blocks. The operand is assigned the SSU version at the top of its stack if the top of its stack is a store or a Λ; otherwise, it is assigned $\perp$.

To recognize that local variables are dead at exits, we assume there is a virtual store to each local variable at each exit of the program unit. Since these virtual stores are first occurrences in the preorder traversal of the post-dominator tree, they are assigned unique SSU versions. Any stores further down in the post-dominator tree that are assigned the same SSU versions are redundant and will be deleted.

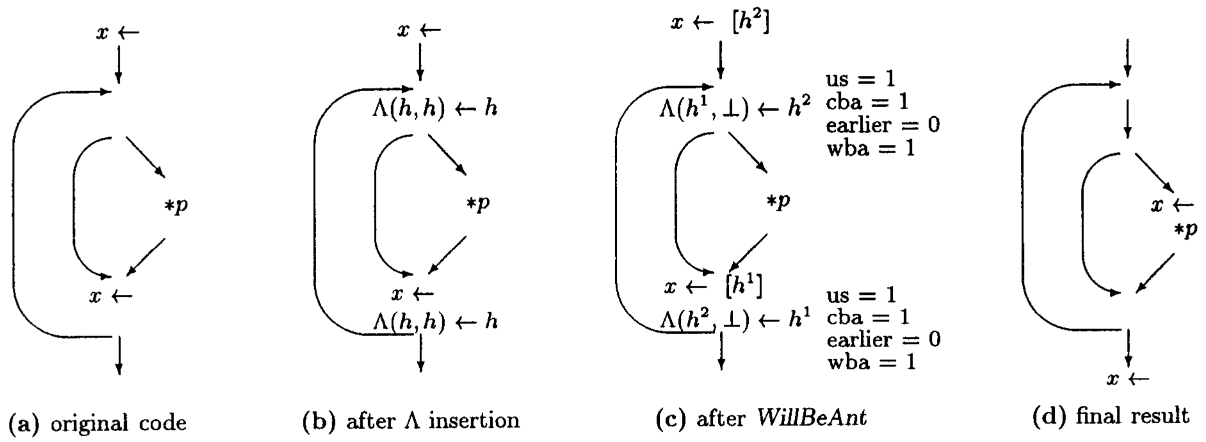
6.3.3 The *UpSafety* Step

One criterion required for PRE to insert a store is that the store be up-safe (*i.e.*, available) at the point of insertion. This step computes up-safety for the Λ's by forward propagation along the edges of the SSU graph. A Λ is up-safe if, in each backward control flow path leading to the procedure entry, another store is encountered before reaching the procedure entry, an aliased load or an aliased store. The propagation algorithm is an exact transposition of the *DownSafety* algorithm in SSAPRE.

To perform speculative store placement, we apply the strategies discussed in Section 5. Figure 11 shows an example where a store is speculatively moved out of the loop.

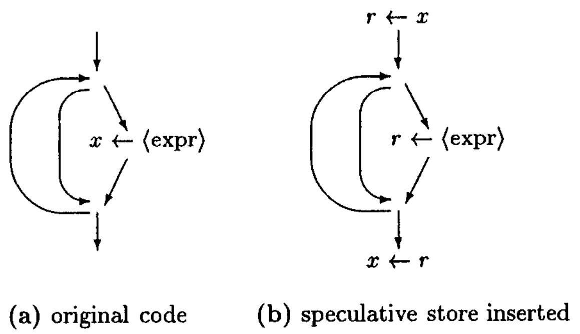
6.3.4 The *WillBeAnt* Step

The *WillBeAnt* step predicts whether the store will be anticipated at each Λ following insertions for PRE. The algorithm again is an exact transposition of the *WillBeAvail* algorithm in SSAPRE. It consists of two backward propagation passes performed sequentially. The first pass computes the *can_be_ant* predicate for each Λ. The second pass works within the region of *can_be_ant* Λ's and compute *earlier*. A



$*p$ is an aliased use of x

Figure 10: Sparse PRE of stores



$x \leftarrow \langle\text{expr}\rangle$
(a) original code
$r \leftarrow x$
$r \leftarrow \langle\text{expr}\rangle$
$x \leftarrow r$
(b) speculative store inserted

Figure 11: Speculative insertion of store

false value of *earlier* implies that the insertion of store cannot be hoisted earlier without introducing unnecessary store redundancy. At the end of the second pass, *will_be_ant* for a Λ is given by:

$$will\_be\_ant = can\_be\_ant \wedge \neg earlier.$$

Figure 10(c) shows the values of *up_safe* (us), *can_be_ant* (cba), *earlier* and *will_be_ant* (wba) for the example at each Λ. The predicate *insert* indicates whether we will perform insertion at a Λ operand. *insert* holds for a Λ operand if and only if the following hold:

- The Λ satisfies *will_be_ant*; and
- the operand is $\perp$ or *has_real_def* is false for the operand and the operand is used by a Λ that does not satisfy *will_be_ant*; *i.e.*, the store is not anticipated at the Λ operand.

6.3.5 The *Finalize* Step

The *Finalize* step in SSUPRE is simpler than the corresponding step in SSAPRE because placement optimization of stores does not require the introduction of temporaries. This step only identifies the stores that will be fully redundant after taking into account the insertions that will be performed. This is done in a preorder traversal of the post-dominator tree of the program. Renaming stacks are not required, because SSU versions have already been assigned. For each store being optimized, we update and use an array *Ant_use* (*cf.* *Avail_def* in SSAPRE) indexed by SSU version to identify stores that are fully redundant.

6.3.6 The *CodeMotion* Step

This last step performs the insertion and deletion of stores to reflect the results of the store placement optimization. The stores inserted always use the pseudo-register as their right hand side, and are of either of the following forms depending on whether the store is direct or indirect:

$$x \leftarrow r \qquad \text{or} \qquad *p \leftarrow r$$

It is necessary to make sure that the value of the pseudo-register r is current at the inserted store. This implies that we need to check if the definitions of r track the definitions of the redundant stores. To do this, we follow the use-def edges in SSA form to get to all the definitions of x that reach the point of store insertion. If the right hand side of a definition is r, the store is simply deleted.[7] If the right hand side is not r, we change it to r, thereby removing the store, which must have been marked redundant by the *Finalize* step:

$$x \leftarrow \langle\text{expr}\rangle \Longrightarrow r \leftarrow \langle\text{expr}\rangle$$

In cases where the inserted store is speculative, it may be necessary to insert a load on the path where the store is not available, so that the pseudo-register will have the right value at the inserted store. In the example of Figure 11, the load $r \leftarrow x$ is inserted at the head of the loop for this reason.

One of our requirements is that the program be maintained in valid SSA form. This implies introducing ϕ's at iterated dominance frontiers and assigning the correct SSA versions for r and x.[8] The current version for r can easily be found by following the use-def edges of x. For x, we assign a new SSA version in each store inserted. Uses of x reached by this inserted store in turn need to be updated to the new version, and can be conveniently handled if def-use chains are also maintained. Instead of maintaining def-use chains, we find it more expedient to perform this task for all affected variables by adding a post-pass to SSUPRE. The post-pass is essentially the renaming step in the SSA construction algorithm, except that rename stacks only need to be maintained for the affected variables.

[7] The right hand side is a pseudo-register if the store was a left occurrence that effected redundancy elimination in the load placement phase.

[8] In the case of indirect stores, the virtual variables have to be maintained in correct SSA versions; see [CCL+96].

benchmark	PRE of loads off A	PRE of loads on B	ratio B/A
099.go	117956223	84137830	0.713
124.m88ksim	25773924	18711578	0.726
126.gcc	326527385	233954451	0.716
129.compress	9334007	6191108	0.663
130.li	41350355	31052102	0.751
132.ijpeg	258354279	227019081	0.879
134.perl	357417768	293678627	0.822
147.vortex	558680056	395136113	0.707
geom. mean			0.744

Table 2: Dynamic counts of *all* loads executed in SPECint95

7 Measurements

We have implemented the register promotion techniques described in this paper in the Silicon Graphics MIPSpro Compilers Release 7.2. In this section, we study the effectiveness of our techniques by compiling the SPECint95 benchmark suite and measuring the resulting dynamic load and store counts when the benchmarks are executed using the training input data. The benchmarks were compiled at the -O2 optimization level with no inlining. Only intra-procedural alias analysis was applied. The measurement data are gathered by simulating the compiled program after register promotion, but before code generation and register allocation. In the simulation, each access to a pseudo-register is not counted as load or store. This is equivalent to assuming that the underlying machine has an infinite number of registers; the assumption allows us to measure the effects of register promotion without confounding effects such as spilling performed by the register allocator.

Our measurement data are organized into two sets. In the first set, we measure the overall effectiveness of the SSAPRE and SSUPRE approaches in removing scalar and indirect loads and stores in the programs. In the second set, we study the relative performance of the different speculative code motion strategies presented in Section 5.

7.1 Overall Performance

Since our register promotion is made up of the PRE of loads and the PRE of stores, we present our data for loads and stores in separate tables. Tables 2 and 3 show the effects of performing PRE of loads and stores respectively on the SPECint95 benchmarks, without speculative code motion. The data shown include both scalar and indirect loads/stores. Column A in the tables shows the number of loads/stores executed in the benchmark programs if register promotion is turned off. Even though register promotion is disabled, non-aliased local variables and compiler-generated temporaries are still assigned pseudo-registers by other parts of the compiler, so the baselines shown in column A represent quite respectable code quality. Column B shows the effects on the number of executed loads and stores when the PRE of loads and the PRE of stores are enabled.

According to Table 2, the PRE of loads reduces the dynamic load counts by an average of 25.6%. In contrast, Table 3 shows the PRE of stores is able to reduce the dynamic store counts by only an average of 1.2%. There are a number of reasons. First, earlier optimization phases have applied the SSA-based dead store elimination algorithm [CFR+91], which efficiently removes all faint and dead stores; the only opportunities left are those exposed by the removal of loads

benchmark	PRE of stores off A	PRE of stores on B	ratio B/A
099.go	23498625	23210822	0.988
124.m88ksim	9174980	9138470	0.996
126.gcc	34206838	33456557	0.978
129.compress	5413281	5274487	0.974
130.li	18589607	18430874	0.991
132.ijpeg	86275191	86274012	1.000
134.perl	141787740	138402068	0.976
147.vortex	148604397	148603779	1.000
geom. mean			0.988

Table 3: Dynamic counts of *all* stores executed in SPECint95

or those due to strictly partial store redundancy. The side effect of earlier loop normalization also moves invariant stores to the end of loops [LLC96]. Second, for aliased variables, there are usually aliased uses around aliased stores, and these uses block movement of the stores. Third, apart from aliased local variables, the other candidates for the PRE of stores are global variables, and they tend to exhibit few store redundancies. Our PRE of stores is performed after the PRE of loads. If the PRE of stores is performed when the PRE of loads is turned off, the resulting dynamic store counts are identical to those in column A, indicating that removing loads is crucial to the removal of stores.

7.2 Speculative Code Motion Strategies

In this section, we study the relative performance of the different speculative code motion strategies we presented in Section 5. Although speculative code motion is applicable to any computation, we concern ourselves only with loads and stores in this paper. In our target architecture, the MIPS R10000, indirect loads and indirect stores cannot be speculated. Thus, we exclude register promotion data for indirect loads and stores from this section, and present data only for scalar loads and stores.

Tables 4 and 5 show the effects of different speculative code motion strategies on the executed scalar load and store counts respectively in the SPECint95 benchmarks. Our baseline is column A, which shows the scalar load and store counts without register promotion. Column B shows the same data when PRE of loads and stores are applied without speculation. Column C shows the result of applying the conservative speculative code motion strategy we presented in Section 5.1. Column D shows the result when we apply the profile-driven speculative code motion strategy described in 5.2 guided by execution profile data. Column E is provided for comparison purposes only; it shows the results if we perform *full* speculation, as defined in Section 5, in the PRE of loads and stores. Only column D in the tables uses profile data.

According to the results given by column C, D and E, full speculation yields the worst performance of the three different speculative code motion strategies. This supports our conviction that it is worth the effort to avoid overdoing speculation. Full speculation yields improvement over no speculation only with loads in **126.gcc** and stores in **134.perl**. When profile data are unavailable, our conservative speculation strategy yields mixed results compared with no speculation, as indicated by comparing columns B and C. There is no effect on a number of the benchmarks. Where there is a change, the effect is biased towards the negative side. Our

benchmark	PRE of loads off A	PRE of loads on B	ratio B/A	conservative speculation C	ratio C/A	profile-driven speculation D	ratio D/A	full speculation E	ratio E/A
099.go	12726299	8440420	0.663	8552367	0.672	8071927	0.634	9444171	0.742
124.m88ksim	3853945	3273909	0.849	3260911	0.846	3228320	0.838	3292380	0.854
126.gcc	49336016	37277650	0.756	37451050	0.759	31580640	0.640	35409194	0.718
129.compress	6535476	3698081	0.566	3698081	0.566	3697881	0.566	3711640	0.568
130.li	14810612	7637328	0.516	7637328	0.516	7637328	0.516	7637328	0.516
132.ijpeg	8368467	3810877	0.455	3810877	0.455	3810877	0.455	4096077	0.489
134.perl	75421556	44879111	0.595	44879460	0.595	44850944	0.595	64817758	0.859
147.vortex	131529172	106031166	0.806	106031170	0.806	104991505	0.798	107055551	0.814
geom. mean			0.637		0.638		0.619		0.680

Table 4: Dynamic counts of *scalar* loads executed in SPECint95 due to the three different speculation strategies

benchmark	PRE *of* stores off A	PRE *of* stores on B	ratio B/A	*conservative* speculation C	ratio C/A	*profile-driven* speculation D	ratio D/A	*full* speculation E	ratio E/A
099.go	6132647	5844844	0.953	5801449	0.946	5811653	0.948	6751996	1.101
124.m88ksim	1122613	1086103	0.967	1086103	0.967	1086103	0.967	1086103	0.967
126.gcc	8960546	8210265	0.916	8539560	0.953	8144287	0.909	8905912	0.994
129.compress	2127310	1988516	0.935	1988516	0.935	1988516	0.935	2002400	0.941
130.li	8471846	8313113	0.981	8313113	0.981	8313113	0.981	8341468	0.985
132.ijpeg	1634572	1633393	0.999	1633393	0.999	1633393	0.999	1633393	0.999
134.perl	42306196	38920524	0.920	38920524	0.920	37309510	0.882	38773931	0.917
147.vortex	115513724	115513106	1.000	115513106	1.000	115513106	1.000	115522935	1.000
geom. mean			0.958		0.962		0.952		0.987

Table 5: Dynamic counts of *scalar* stores executed in SPECint95 due to the three different speculation strategies

conservative speculative code motion strategy can increase the executed operation count if some operations inserted in loop headers would not have been executed in the absence of speculation. In the case of the SPECint95 benchmark suite, this situation seems to arise more often than not.

When profile data are available, our profile-driven speculative code motion strategy consistently yields the best results. This outcome is indicated in column D, which shows the best number for each benchmark among all the columns in Tables 4 and 5. The data show that our profile-driven speculation strategy is successful in making use of execution frequency information in avoiding over-speculation by speculating only in cases where it is certain of improvement. In programs with complicated control flow like **126.gcc**, our profile-driven speculation yields greater improvements. Since inlining augments control flow, we can expect even better results if the benchmarks are compiled with inlining. Overall, profile-driven speculation contributes to 2% further reduction in dynamic load counts and 0.5% further reduction in dynamic store counts. When profile-driven speculation is applied to other types of operations that can be speculated, we can expect similar reduction in dynamic counts for those operations. Given the current trend toward hardware support for more types of instructions that can be speculated, we can expect our profile-driven speculation algorithm to play a greater role in improving program performance for newer generations of processors.

8 Conclusion

In this paper, we have presented a pragmatic approach to register promotion by modeling the optimization as two separate problems: the PRE of loads and the PRE of stores. Both of these problems can be solved through a sparse approach to PRE. Since the PRE of loads uses the same algorithm as the PRE of expressions, it can be integrated into an existing implementation of SSAPRE with minimal effort and little impact on compilation time. The PRE of stores uses a different algorithm, SSUPRE, which is the dual of SSAPRE, and is performed after the PRE of loads, taking advantage of the loads' having been converted into pseudo-register references so that there are fewer barriers to the movement of stores. SSUPRE is not as efficient as SSAPRE because the SSA form of its input does not directly facilitate the construction of SSU form. Since register promotion is relevant only to aliased variables and global variables, the number of candidates in each program unit is usually not large. Therefore a sparse, per-variable approach to the problem is justified. In contrast, a bit-vector-based approach takes advantage of the parallelism inherent in bit vector operations, but incurs some larger fixed cost in initializing and operating on the bit vectors over the entire control flow graph. As a result, we have seen very little degradation in compilation time with our sparse approach compared to a bit-vector-based implementation that precedes this work.

We have presented techniques to effect speculative code motion in our sparse PRE framework. In particular, the profile-driven speculation technique enables us to use profile data to control the amount of speculation. These techniques could not be efficiently applied in a bit-vector-based approach to PRE. Our measurement data on the SPECint95 benchmark suite demonstrate that substantial reduction in load counts is possible by applying our techniques to aliased variables, global variables and indirectly accessed memory locations. There is not a large reduction in store counts due to earlier optimizations in the compiler. Our study of the different speculative code motion strategies shows that the profile-driven speculative code motion algorithm is most promising in improving program performance over what can be achieved by partial redundancy elimination.

References

[BG97] R. Bodík and R. Gupta. Partial dead code elimination using slicing transformations. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 159–170, June 1997.

[Bri92] P. Briggs. Rematerialization. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 311–321, June 1992.

[CAC+81] G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, January 1981.

[CCK+97] 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, June 1997.

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

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

[CH90] F. Chow and J. Hennessy. The priority-based coloring approach to register allocation. *ACM Trans. on Programming Languages and Systems*, 12(4):501–536, October 1990.

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

[CL97] K. Cooper and J. Lu. Register promotion in C programs. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 308–319, June 1997.

[Dha88] D. Dhamdhere. Register assignment using code placement techniques. *Journal of Computer Languages*, 13(2):75–93, 1988.

[Dha90] D. Dhamdhere. A usually linear algorithm for register assignment using edge placement of load and store instructions. *Journal of Computer Languages*, 15(2):83–94, 1990.

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

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

[FKCX94] L. Feigen, D. Klappholz, R. Casazza, and X. Xue. The revival transformation. In *Conference Record of the Twenty First ACM Symposium on Principles of Programming Languages*, pages 147–158, January 1994.

[GBF97a] R. Gupta, D. Berson, and J. Fang. Path profile guided partial dead code elimination using predication. In *Proceedings of the Fifth International Conference on Parallel Architectures and Compilation Techniques*, pages 102–113, November 1997.

[GBF97b] R. Gupta, D. Berson, and J. Fang. Resource-sensitive profile-directed data flow analysis for code optimization. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 358–368, December 1997.

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

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

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

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

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

[TWL+91] S. Tjiang, M. Wolf, M. Lam, K. Pieper, and J. Hennessy. Integrating scalar optimization and parallelization. In *Proc. 4th International Workshop on Languages and Compilers for Parallel Computing*, pages 137–151, August 1991.

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