Register Pressure Sensitive Redundancy Elimination*

Rajiv Gupta and Rastislav Bodík

Dept. of Computer Science, Univ. of Pittsburgh, Pittsburgh, PA 15260, USA

Abstract. *Redundancy elimination* optimizations avoid repeated computation of the same value by computing the value once, saving it in a temporary, and reusing the value from the temporary when it is needed again. Examples of redundancy elimination optimizations include common subexpression elimination, loop invariant code motion and partial redundancy elimination. We demonstrate that the introduction of temporaries to save computed values can result in a significant increase in register pressure. An increase in register pressure may in turn trigger generation of spill code which can more than offset the gains derived from redundancy elimination. While current techniques *minimize* increases in register pressure, to avoid spill code generation it is instead necessary to ensure that register pressure *does not exceed* the number of available registers.

In this paper we develop a redundancy elimination algorithm that is sensitive to register pressure: our novel technique first sets upper limits on allowed register pressure and then performs redundancy elimination within these limits. By setting strict register pressure limits for frequently executed (hot) blocks and higher limits for infrequently executed (cold) blocks, our algorithm permits trade-off between redundancy removal from hot blocks at the expense of introducing spill code in cold blocks. In addition, the program profile is also used to prioritize optimizable expressions; when not enough registers are available, the most profitable redundancies are removed first. To increase redundancy elimination within the allowed register pressure, our algorithm lowers the pressure with two new program transformation techniques: (a) whenever possible, we avoid inserting a temporary and instead access the reused value from existing variables, which reduces the life time of the temporary beyond existing live-range optimal algorithms; and (b) the live ranges of variables referenced by the expressions are shortened by combining expression hoisting with assignment sinking.

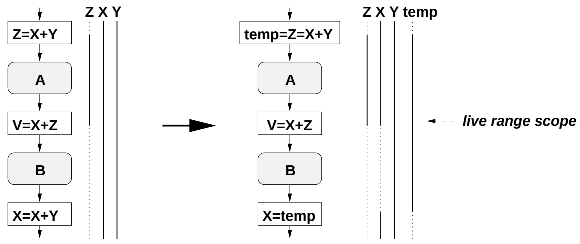
Keywords - data flow analysis, code optimization, partial redundancy elimination, partial dead code elimination, code motion, register pressure, spill code.

* Supported in part by NSF grants CCR-9808590, EIA-9806525 and a grant from Intel Corporation to the University of Pittsburgh.

1 Introduction

Redundancy elimination is an important commonly implemented optimization. Loop invariant code motion (LICM) eliminates from loops statements that compute the same value in each loop iteration. Global common subexpression elimination (CSE) eliminates an expression that is preceded by an identical computation along all incoming paths. Finally, partial redundancy elimination (PRE) subsumes LICM and CSE by eliminating redundancy from instructions that are redundant along only a subset of incoming paths [1, 6, 7, 9, 13, 14, 16, 17, 19]. Since PRE is the most general redundancy elimination, the focus of this paper is on developing an improved PRE algorithm.

PRE algorithms avoid repeated computation of the same value by computing the value once, saving it in a temporary, and reusing the value from the temporary when it is needed again. In the code below, the recomputation of X+Y is optimized away by remembering its value in the temporary temp.



However, an additional register must be allocated for temp, increasing the register pressure in block A. This increase in register pressure may result in generation of memory intensive spill code, which can more than offset the gains derived from redundancy elimination. While it is widely believed that PRE impacts register pressure, and some existing algorithms even attempt to minimize register pressure increase [15, 17], the effects of PRE algorithms on register pressure have not been evaluated.

Let us consider the impact of *lazy code motion* [17] PRE on register pressure (although [15] is the best known algorithm, our implementation currently supports [17]). The algorithm attempts to reduce the increase in register pressure by minimizing the lengths of live ranges for the temporaries introduced to hold values of redundant expressions. Figure 1a illustrates the effect of lazy code motion on register pressure. Plotted in this graph is the average register pressure (y-axis) for all basic blocks that have a given execution frequency (x-axis). The top curve shows the running average of register pressure (i.e., the number of live ranges crossing the basic block entry) prior to the application of PRE. The middle and the bottom curves give the average number of live ranges added and removed due to PRE. For example, block B above has one added and one removed live range (temp vs. X). The overall change in register pressure is the difference of the middle and the bottom curves.

This graph documents that PRE causes a significant (approx. 25%) average increase in register pressure. Most importantly, the increase is not limited to low frequency basic blocks; the resulting spill loads and stores might slow down important, hot basic blocks. The graph reflects only redundancy of expressions that are lexically identical; if our experiments used value numbering techniques to discover additional redundancies [4, 6, 20], significantly higher increases in register pressure would be observed [2, 6].

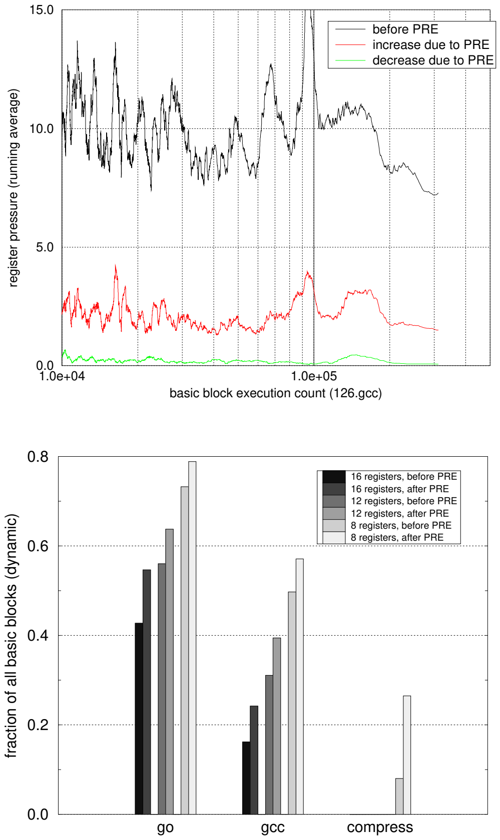


Fig. 1. Effect of lazy code motion PRE on register pressure: (a) Each point corresponds to the average number of live ranges (y) of all basic blocks with a given execution frequency (x). There were about 82,000 executed basic blocks in program 126.gcc from SPEC95. (b) Bars plot the number of executed basic blocks whose register pressure exceeded given limit before and after PRE.

Register pressure changes are harmful only if spill code is generated as a result. Our second experiment aimed to determine whether PRE may indeed increase the pressure above the number of commonly available physical registers, triggering spill code generation. We measured the dynamic number of executed basic blocks whose pressure was above 16, 12, and 8 registers, both before and after PRE. As shown in Figure 1b, the increase in the fraction of basic blocks that exceeded these limits is significant (5–10%). This increase translates to a corresponding increase of basic blocks that will execute some spill code after PRE.

The second experiment exposes the practical inadequacy of the current techniques [15, 17], in which minimizing register pressure comes only second to the primary goal of maximizing redundancy removal. Figure 1b convinces us that removing *all* redundancy at any cost may be harmful as it may still cause a significant spill code. We argue that an effective PRE must instead consider redundancy removal and register pressure in balance: when no more registers are available, remaining PRE opportunities must be sacrificed.

Another important observation behind our algorithm is that some PRE opportunities may decrease register pressure (for example, see the live range of X in block B in our example). Therefore, when selecting the subset of PRE opportunities to optimize, these expressions can be used to enable optimization of more important expressions. Furthermore, since numerous basic blocks exceed available register resources already prior to PRE, in addition to minimizing register pressure increases, these pressure releaving expressions might eliminate spill code that was present prior to PRE.

In order to address the above issues we have developed a new register pressure sensitive redundancy elimination algorithm with the following characteristics:

- Our algorithm sets upper limits on allowed register pressure and attempts to maximize elimination of dynamically observed redundancy within these limits. While strict register pressure limits are set for frequently executed (hot) blocks, higher register pressures are permitted in infrequently executed (cold) blocks. Therefore, insufficient registers in cold blocks do not prevent optimization in hot blocks.
- Since under limited register resources our algorithm may exploit only a subset of PRE opportunities, estimates of dynamic PRE benefits are used to give higher priority to the removal of redundancies that yield the most benefit. Moreover, by applying PRE in situations where a reduction in register pressure results, we further enable exploitation of additional PRE opportunities.
- To minimize increase in register pressure, the life time of a *temporary* is reduced beyond existing live-range optimal algorithms [17] by accessing the value from existing program variables that already contain the value. Only when the reused value is not by available in any existing variable, a temporary is introduced to carry the value. In comparison with [17], we reduce temporary live ranges on all paths, rather than only on paths where the value was previously not available. In our example, we would initialize temp from Z after block A, rather than before it, avoiding pressure increase in block A.
- Further reductions in register pressure can be achieved by minimizing the live ranges of *program variables* referenced by optimized expressions through a combination of expression hoisting with assignment sinking. In our example, sinking Z=X+Y to below block A would reduce the live range of Z, without extending live ranges of X and Y. We describe how sinking and hoisting can be used for register pressure sensitive PRE.

2 Register Pressure Guided PRE

We begin by providing an overview of our algorithm, then we discuss the critical steps of the algorithm in greater detail. As shown in Figure 2, our algorithm begins by computing the register pressure and setting an upper limit on the register pressure for each basic block based upon profile information. The register pressure is computed as the maximum number of live variables at any point in the basic block. If the register pressure of a block is already equal to or higher than this limit, no further increases would be allowed. On the other hand if the pressure is lower than the limit, increases up to the limit are allowed.

To uncover opportunities for PRE, anticipability and availability analysis is performed as described in [1]. The dynamic benefit of optimization opportunities associated with each lexical expression is computed using profile data [1]. PRE within the limits of allowed increases in register pressure is performed using a greedy algorithm which prioritizes the expressions according to their benefits and applies PRE to the expression with highest benefit first.

```
for each basic block B
    Determine register pressure RP(B)
    Set register pressure limit, LRP(B), based on freq(B)
end for
for each lexical expression exp
    Perform availability and anticipability analysis on exp
    Compute dynamic benefit of applying PRE to exp
end for
while all expressions have not been considered
    Select the unoptimized expression, uexp,
       with the highest dynamic benefit
    for each basic block B involved in PRE of uexp
        Compute uexp's PRE caused register pressure changes, δRP(B)
        if δRP(B) + RP(B) > LRP(B) then
           Attempt to reduce register pressure of B by
             LRP(B)-δRP(B) + RP(B) by hoisting
             of expressions out of B
           if register pressure is not adequately reduced then
              Inhibit expression hoisting of uexp through B
           end if
        end if
    end for
    Update availability and anticipability information to
        reflect inhibited expression hoisting
    Perform maximal PRE of uexp within register pressure limits
end while
Perform further PRE through assignment sinking
```

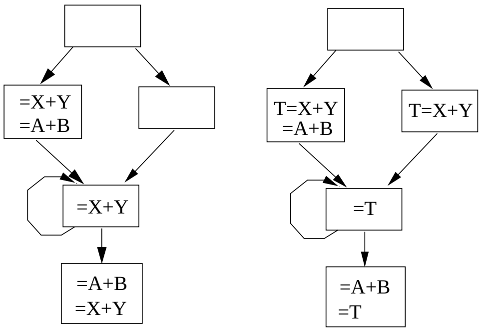
Fig. 2. Algorithm for register pressure guided PRE.

Once an expression has been selected for optimization, we compute the changes in register pressure that would result if all of the uncovered PRE opportunities involving that expression are exploited. If changes reflect that for some blocks the increase in register pressure would result in increased spill code, then we attempt to reduce register pressure in those blocks. The reductions are achieved by hoisting expressions out of the blocks with at least one operand that is no longer live after its use by the expression, since the live range of such an operand would be shortened upon hoisting. In other words expression hoisting would continue to shorten the live range of the operand variable until another use of the same operand is encountered. Of course it may not always be possible to achieve the desired reduction in register pressure for a basic block, in which case we inhibit the hoisting of the expression being optimized through that block. For the purpose of analysis, a point at which hoisting is inhibited is essentially viewed as a kill point for the expression. By doing so we inhibit the increase in the register pressure for the block. The affect of inhibiting code hoisting through the block is to disable those opportunities for PRE, involving the current expression, that are enabled by the inhibited code motion. We update the availability and anticipability information to reflect the killing affects of inhibited code motion by treating them as kill points for the current expression. Based upon updated anticipability information and availability information we perform the PRE transformation. In this way we perform PRE to the extent that is allowed by register pressure limits. The above process is similarly repeated for other expressions.

2.1 Using profile data

Profile information serves two important functions in our algorithm. First we use profile information to compute the dynamic benefit of redundancy removal for various expressions. The benefits are used to *selectively* apply PRE during which higher priority is given to redundancy elimination opportunities that result in the most benefit. The second use of profiling is in setting register pressure limits. In particular, strict register pressure limits are set on hot basic blocks while higher register pressure limits are set for cold blocks. In this way we are able to achieve redundancy removal in hot blocks at the expense of allowing spill code in cold blocks.

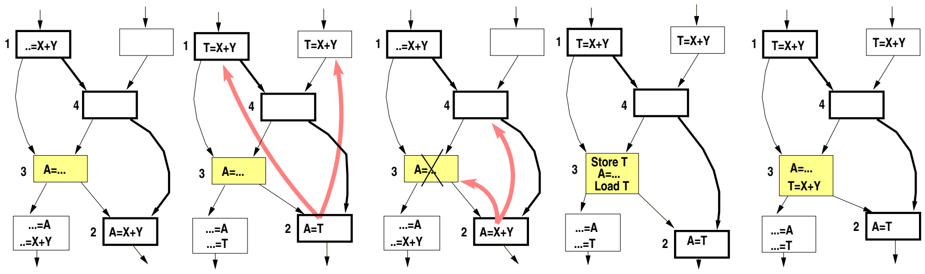
Setting priorities. Consider the example in Figure 3a in which redundancy exists both in the evaluations of X+Y and A+B. Profile information indicates that X+Y, which is repeatedly computed in the loop, should have priority over A+B. Let us assume that variables X, Y, A and B are live throughout the program and a single register is free. This register can be used to hold the value of T and thus removal of redundancy of X+Y is achieved, as shown in Figure 3b, without generation of additional spill code. Redundancy of A+B is not removed as there are no more free registers.



(a) Original Code. (b) After PRE.

Fig. 3. Selective PRE: optimize high-benefit redundancies in a greedy fashion.

Setting register pressure limits. Next consider the example in Figure 4a. Assuming that the execution frequency of the highlighted path from block 1 to block 2 is very high, we would like to eliminate the redundancy involving X+Y from this path. Furthermore, let us assume that block 3 has a low execution frequency but high register pressure (A is live in block 3 but dead in blocks 1 and 2). If PRE is applied as shown in Figure 4b, the register pressure in block 3 would further increase. On the other hand, if we prevent register-pressure increase in block 3, and thus disable hoisting of X+Y above node 4, PRE along the highlighted path would not be achieved (Figure 4c). Given that the execution frequency of block 3 is low and hence spill code does no harm, it would be preferable to set a higher limit for block 3 and thus enable PRE along the frequently executed path. The register allocator would then either spill the value of T in block 3 as shown in Figure 4d or rematerialize [5] X+Y as shown in Figure 4e. Thus, by setting a higher register pressure limit for cold blocks we can trade-off PRE that benefits hot blocks at the cost of poorer performance for cold blocks.



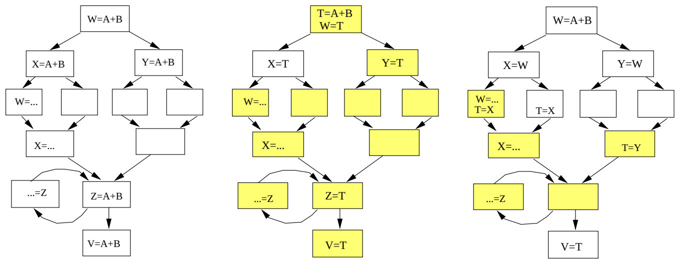
(a) Before PRE. (b) After PRE (hoisting enabled). (c) Hoisting blocked. (d) Spill Code Generation. (e) Rematerialization.

Fig. 4. Allowing register-pressure increases in cold blocks enables PRE in hot blocks.

2.2 Computing register pressure changes

PRE affects register pressure in two ways. First, because a temporary is introduced to carry the value of the redundant expression, the portions of the program over which the temporary is live experience an increase in register pressure. Second, because the redundant expression itself is hoisted, the live range of a variable referenced in the expression is shortened if the expression represents the last use of the variable, causing a decrease in register pressure. By determining the actual changes in live ranges for the temporary and for the expression operands, we compute the changes in register pressure that would result from PRE of the expression.

Live range of the temporary. Existing live-range optimal algorithms initialize the temporary at the point where the expression is computed for the first time, i.e., X:=A+B is replaced with T:=X:=A+B. The temporary T is then used to provide the value at points where redundant instances of A+B were removed. The temporary is however unnecessary at points where the value of A+B is still held by the left-hand-side variable X. Our approach is to further reduce the live range of the temporary by obtaining the value of the expression from other variables that may also contain its value.



(a) Before PRE. (b) After lazy code motion PRE. (c) After register sensitive PRE.

Fig. 5. Introducing temporary to hold an expression's value.

Consider the example in Figure 5a which after traditional PRE results in the code shown in Figure 5b. As we can see the live range of the temporary T introduced in this case extends over the entire code segment (shown by shaded blocks). We can reduce the increase in register pressure by narrowing this live range as shown in Figure 5c. The reduction is achieved by exploiting the observation that in parts of the code the value of the expression is already available in existing variables. Note that in this approach copy assignments are introduced to initialize the temporary T at points where the live range of T begins. These copy assignments are typically eliminated during register allocation.

Next we present describe the computation of a new optimal live range for PRE which essentially works like the computation of live range in lazy code motion but delays the temporaries. This algorithm performs forward propagation of assignments which compute the expression under consideration. At each program point where at least one such assignment is available along all paths, the value of the expression can be accessed from the variable on the left hand side of that assignment and thus there is no need for a new temporary to hold the expression's value. The copy assignments are introduced at the latest points where the expression's value is available through an existing variable. In the analysis below, $\mathsf{NAVAIL}_{op(x,y)}(n)(\mathsf{XAVAIL}_{op(x,y)}(n))$ represents the must-availability of all program assignments which compute the lexical expression op(x,y) at the entry(exit) of node n (i.e., must-availability for all assignments of the form v:=op(x,y) is computed).

$$\mathsf{NAVAIL}_{op(x,y)}(n) = \sqcap_{p \in Pred(n)} \mathsf{XAVAIL}_{op(x,y)}(n) - \mathsf{DEAD}_{op(x,y)}(n)$$

$$\mathsf{XAVAIL}_{op(x,y)}(n) = (\mathsf{NAVAIL}_{op(x,y)}(n) - \mathsf{KILL}_{op(x,y)}(n)) \sqcup \mathsf{GEN}_{op(x,y)}(n)$$

$$\mathsf{DEAD}_{op(x,y)}(n) = \{stat.\ v := op(x,y) :\ v\ is\ dead\ at\ n's\ entry\}$$

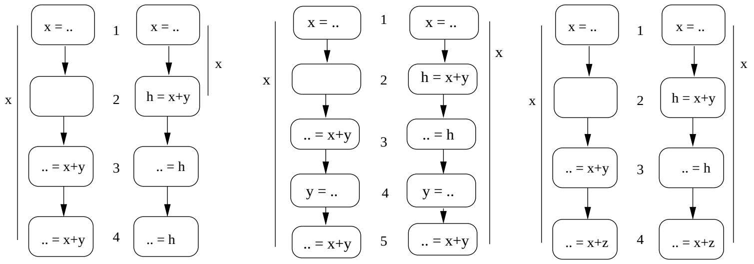
$$\mathsf{KILL}_{op(x,y)}(n) = \{stat.\ v := op(x,y) :\ n\ defines\ x, y\ or\ v\}$$

$$\mathsf{GEN}_{op(x,y)}(n) = \{stat.\ v := op(x,y) :\ n\ computes\ v := op(x,y)\}$$

The above analysis will identify for the example in Figure 5c that the value of the expression A+B is available in variables W, Y and/or X is certain program regions while in others a new temporary T is required. The initialization of T through copying is performed at node exits that are the latest points at which the value of the expression is available in an existing variable (i.e., the value is not available at one of the sucessors of the node).

Live ranges of referenced variables. To track the changes in the live range of a referenced variable x when PRE for expression x+y is carried out, we must take into account the effect of hoisting x+y on x's liveness. We develop the notion of *PRE-liveness* in which the liveness of a variable x is computed in relation to an expression x+y which is being subjected to hoisting. Under this notion of liveness, x is live for x+y at a program point n if and only if even after PRE has been able to hoist x+y above n, x is still live at n. On the other hand if x is not live at n after PRE has hoisted x+y above n, then x is considered to be dead for x+y.

The notion of *PRE-liveness* is illustrated in Figure 6. Consider the situations in which multiple (say two) evaluations of x+y are present on a path and the latest evaluation of x+y represents the last use of x. If both of the evaluations of x+y are hoisted by PRE as shown in Figure 6a, then the live range of x will be shortened. Thus, x is considered to be dead for x+y at the exit of node 2 in Figure 6a. If the last evaluation of x+y cannot be hoisted due to a definition of y, then there is no change in the length of the live range of x. Thus, in Figure 6b x is considered to be live for x+y at the exit of node 2. If after the use of x in x+y, there is another use of x by an expression which is lexically different from x+y, then the length of x's live range will remain the same. Conservatively we consider a variable to be PRE-live if there is at least one path along which it is PRE-live. In Figure 6c x is considered to be live for x+y at the exit of node 2.



(a) Hoisting of all uses of x in x+y and PRE. (b) Blocked use of x in x+y. (c) Hoisting of uses of x in x+y and PRE.

Fig. 6. PRE-liveness of x: The live ranges after PRE are the PRE live ranges.

Definition: Variable x is *PRE-live* wrt expression op(x,y) at point n if and only if: (i) there exists a path from n to an evaluation of op(x,y) which does not contain any redefinition of x but contains a redefinition of y (see Figure 6b); or (ii) there exists a path from n to an evaluation of an expression op(x,-) (an expression which uses x and is lexically different from op(x,y)) which does not contain any redefinition of x (see Figure 6c).

Next we present the data flow equations for computing the PRE-liveness information. Since both conditions in the above definition require backward flow analysis, we can evaluate them simultaneously as a single data flow problem. For this purpose the data flow solution at a point is represented by one of three values $\top$, USE and $\perp$, where $\top \sqsubset \mathsf{USE} \sqsubset \perp$. The join operator, $\sqcap$, used during data flow is also defined below. Initially the data flow value at a node is $\top$ indicating that no use of the current value of x exists along any path. The value is lowered to $\perp$ if a use by op(x,-) is found indicating the value is to be considered live according to condition (ii) of the definition. The value is lowered from $\top$ to USE if a use of x's value by op(x,y) is found. For points to which op(x,y) cannot be hoisted because of a definition of y, the data flow value is lowered from USE to $\perp$ indicating that the current value of x is live for op(x,y) according to condition (i) of the definition. The PRE-liveness value $\mathsf{NPRELIVE}_x^{op(x,y)}(n)$ at the entry of each node n is true if the data flow solution $\mathsf{NEXP}_x^{op(x,y)}(n)$ is $\perp$; otherwise it is false. The liveness at exit, $\mathsf{XPRELIVE}_x^{op(x,y)}(n)$, is similarly computed from $\mathsf{XEXP}_x^{op(x,y)}(n)$.

Liveness analysis for x wrt expression $op(x,y)$ at node n for any path from n to end.

$\top$	x is not used
USE	x is used only by occurrences of op(x,y) whose hoisting is not blocked by y's definition
$\perp$	x is used by op(x,-) or by occurrences of op(x,y) whose hoisting is blocked by y's definition

$\sqcap$	$\top$	USE	$\perp$
$\top$	$\top$	USE	$\perp$
USE	USE	USE	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$

$$\mathsf{NEXP}_x^{op(x,y)}(n) = \begin{cases} \perp & if\ (\mathsf{XEXP}_x^{op(x,y)}(n) = \mathsf{USE} \wedge n\ defines\ y) \\ & \vee\ (op(x,-) \in n) \\ \mathsf{USE} & if\ op(x,y)\ \in\ n \\ \mathsf{XEXP}_x^{op(x,y)}(n) & otherwise \end{cases}$$

$$\mathsf{XEXP}_x^{op(x,y)}(n) = \begin{cases} \top & if\ n = exit \\ \sqcap_{w \in Succ(n)} \mathsf{NEXP}_x^{op(x,y)}(w) & otherwise \end{cases}$$

$$\mathsf{XPRELIVE}_x^{op(x,y)}(n) = (\mathsf{XEXP}_x^{op(x,y)}(n) = \bot)$$

$$\mathsf{NPRELIVE}_x^{op(x,y)}(n) = (\mathsf{NEXP}_x^{op(x,y)}(n) = \bot)$$

Overall register pressure changes. The overall change in register pressure at the entry(exit) of node n due to PRE of op(x,y), denoted by $\delta\mathsf{NRP}_{op(x,y)}(n)$ ($\delta\mathsf{XRP}_{op(x,y)}(n)$), is computed from the changes in register pressure due to operands (x,y) and the temporary introduced to save the value of op(x,y). The change in register pressure at node n's entry(exit) due to operands of op(x,y), given by $\delta\mathsf{NOPRP}_{op(x,y)}(n)$ ($\delta\mathsf{XOPRP}_{op(x,y)}(n)$), is computed from the PRE-liveness information. The change in register pressure due to a temporary at node n's entry(exit), given by $\delta\mathsf{NTRP}_{op(x,y)}(n)$ ($\delta\mathsf{XTRP}_{op(x,y)}(n)$), is computed from must-availability of assignments that compute op(x,y). Note that in the algorithm of Figure 2, the major step was the computation of register pressure changes.

$$\delta\mathsf{XRP}_{op(x,y)}(n) = \delta\mathsf{XOPRP}_{op(x,y)}(n) + \delta\mathsf{XTRP}_{op(x,y)}(n)$$

$$\delta\mathsf{NRP}_{op(x,y)}(n) = \delta\mathsf{NOPRP}_{op(x,y)}(n) + \delta\mathsf{NTRP}_{op(x,y)}(n)$$

$$\delta\mathsf{XOPRP}_{op(x,y)}(n) = \begin{cases} 0 \quad if\ \mathsf{XPRELIVE}_x^{op(x,y)}(n) \wedge \mathsf{XPRELIVE}_y^{op(x,y)}(n) \\ -1\ elseif\ \mathsf{XPRELIVE}_x^{op(x,y)}(n) \vee \mathsf{XPRELIVE}_y^{op(x,y)}(n) \\ -2\ otherwise \end{cases}$$

$$\delta\mathsf{NOPRP}_{op(x,y)}(n) = \begin{cases} 0 \quad if\ \mathsf{NPRELIVE}_x^{op(x,y)}(n) \wedge \mathsf{NPRELIVE}_y^{op(x,y)}(n) \\ -1\ elseif\ \mathsf{NPRELIVE}_x^{op(x,y)}(n) \vee \mathsf{NPRELIVE}_y^{op(x,y)}(n) \\ -2\ otherwise \end{cases}$$

$$\delta\mathsf{NTRP}_{op(x,y)}(n) = \begin{cases} 1\ if\ \mathsf{NAVAIL}_{op(x,y)}(n) = false\ for\ all\ v := op(x,y) \\ 0\ otherwise \end{cases}$$

$$\delta\mathsf{XTRP}_{op(x,y)}(n) = \begin{cases} 1\ if\ \exists\ s \in Succ(n)\ st\ \mathsf{NAVAIL}op(x,y)(s) = false \\ \quad for\ all\ v := op(x,y) \\ 0\ otherwise \end{cases}$$

The results of applying the above analysis techniques to an example are shown in Figure 7. As we can see, the application of PRE to expression x+y by hoisting results in reduction of register pressure initially. Once the expression is hoisted above uses of x and y the register pressure starts to increase. Assuming the register pressure limits are set such that no increase in register pressure is allowed, the code resulting after PRE is shown in Figure 7b. While PRE due to computation of x+y within the loop and following the loop is removed, still PRE remains due to computation of x+y in the assignment a=x+y. The latter redundancy is not removed because hoisting is disabled to prevent an increase in register pressure.

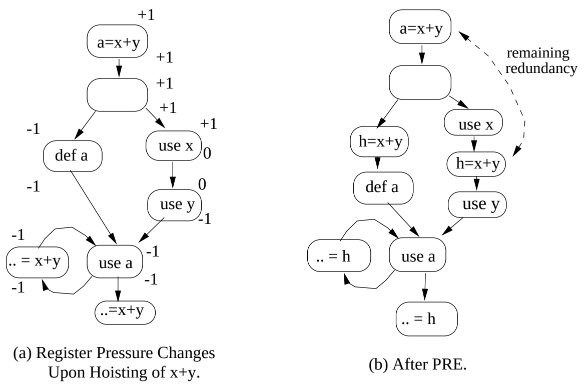


Fig. 7. Example of register pressure changes of operands due to PRE.

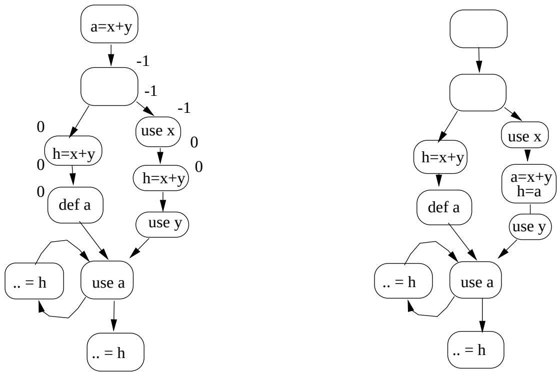
3 Register Pressure and Assignment Motion

In this section we show how the PRE algorithm based upon expression hoisting can be further enhanced through assignment motion.

PRE through assignment sinking. As the example in Figure 7 illustrates, after the algorithm of the preceding section has been applied, redundancy may still remain. It may be possible to remove all or part of the remaining redundancy through assignment sinking. In particular if we can sink an assignment that computes some expression op(x,y) to another computation of op(x,y), then the latter computation can be eliminated. We propose that after applying the presented algorithm, a separate phase may be used to remove as much redundancy as possible through assignment sinking. In order to ensure that sinking does not increase register pressure, we may apply sinking only to the extent that it does not cause any increase in register pressure. Furthermore, since assignment sinking may enable partial dead code elimination [3, 11, 18], it can also be performed at the same time.

Given an assignment, the sinking algorithm consists of three major steps: computing register pressure changes, delayability analysis for determining legality of sinking, and insertion point determination. To compute register pressure changes we observe that if an assignment is subjected to sinking, the length of the live range associated with the variable defined by the statement decreases. The changes in the length of the live range associated with a variable used by the *rhs*-expression in the assignment can be predicted using traditional global liveness information for that variable. If the operand variable is not live prior to sinking, then the live range for the operand increases with sinking. On the other hand if the operand variable is live prior to sinking, the length of the live range remains unaffected as long as at the points through which sinking proceeds variable continues to be live. If a partially dead assignment sinks to a point at which it is fully dead, then no change in register pressure occurs at that point since the assignment would neither be inserted at that point nor sunk any further. The delayability analysis is forward analysis that determines points to which an assignment can be sunk. In this analysis sinking is blocked at appropriate program points to prevent register pressure increase. Finally the assignment being sunk is inserted at a node's entry if it can be delayed to its entry but not its exit. It is inserted at a node's exit if it can be delayed to its exit but not to the entries of one of its successors. In both of the above cases insertion is only needed if the variable that the statement defines is live.

Let us illustrate the use of sinking for performing PRE to the example from the preceding section where some PRE had already been applied through hoisting of x+y while some remains. Figure 8a shows the changes in register pressure for sinking a=x+y. The sinking of a=x+y yields the code in Figure 8b in which the remainder of the redundancy is removed. Thus, PRE is achieved without any increase in register pressure through a combination of hoisting and sinking, while neither by itself could have enabled PRE without an increase in register pressure.



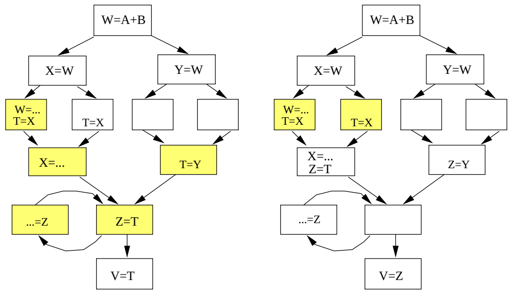
(a) Register Pressure Changes due to Sinking of a=x+y. (b) After PRE & PDE through Sinking.

Fig. 8. Example.

Live range reduction through assignment hoisting. In some situations assignment hoisting can be used to further reduce the live range of the temporary. In the example of Figure 5a, by performing hoisting of loop invariant assignment instead of simply hoisting loop invariant expressions out of loops. As shown in Figure 9b the live range of temporary T is further shortened if the loop invariant assignment Z=A+B is hoisted out of the loop, instead of simply hoisting expression A+B out of the loop as was done in Figure 9a.

4 Concluding Remarks

In this paper we demonstrated the inadequacy of the current PRE techniques [15, 17] which simply try to minimize the increase in register pressure. Minimization



(a) After register sensitive PRE. (b) Allowing Assignment Hoisting.

Fig. 9. Assignment hoisting.

of the increase in register pressure can still lead to a significant increase in spill code. Therefore we proposed an approach to PRE which sets upper limits on allowed register pressures based upon profile information and within the constraints of these limits performs redundancy elimination. In an attempt to maximize redundancy elimination we try to minimize the increase in the register pressure using two new techniques. The first technique reduces the range over which a new temporary is introduced beyond existing live range optimal [17] techniques. The second technique uses assignment sinking to perform PRE that was not achieved by hoisting alone.

Finally we would like to point out that code reordering performed during instruction scheduling also effects register pressure. Our algorithm can be easily extended to perform such code reordering to lower register pressure. However, for such code reordering to be consistent with good instruction scheduling decisions, it is important to consider pressure on functional unit resources. In [12, 10] we presented techniques for code reordering, including for PRE and PDE, that are sensitive to functional unit pressure and consistent with instruction scheduling decisions. Those techniques can be combined with the algorithms presented in this paper.

References

1. R. Bodik, R. Gupta and M.L. Soffa, "Complete Removal of Redundant Expressions," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1-14, Montreal, Canada, June 1998.
2. R. Bodik, R. Gupta and M.L. Soffa, "Load-Reuse Analysis: Design and Evaluation," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, Georgia, May 1999.
3. R. Bodik and R. Gupta, "Partial Dead Code Elimination using Slicing Transformations," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 159-170, Las Vegas, Nevada, June 1997.
4. R. Bodik and S. Anik, "Path-Sensitive Value-Flow Analysis," *25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Diego, California, January 1998.

5. P. Briggs, K.D. Cooper, and L. Torczon, "Rematerialization," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 311-321, June 1992.
6. P. Briggs and K.D. Cooper, "Effective Partial Redundancy Elimination," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 159-170, June 1994.
7. F. Chow, S. Chan, R. Kennedy, S-M. Liu, R. Lo, and P. Tu, "A New Algorithm for Partial Redundancy Elimination based upon SSA Form," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 273-286, Las Vegas, Nevada, June 1997.
8. C. Click, "Global Code Motion Global Value Numbering," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 246-257, La Jolla, CA, June 1995.
9. D.M. Dhamdhere, "Practical Adaptation of Global Optimization Algorithm of Morel and Renvoise," *ACM Transactions on Programming Languages*, 13(2):291-294, 1991.
10. R. Gupta, "A Code Motion Framework for Global Instruction Scheduling," *International Conference on Compiler Construction*, LNCS 1383, Springer Verlag, pages 219-233, Lisbon, Portugal, March 1998.
11. R. Gupta, D. Berson, and J.Z. Fang, "Path Profile Guided Partial Dead Code Elimination using Predication," *International Conference on Parallel Architectures and Compilation Techniques*, pages 102-115, San Francisco, California, November 1997.
12. R. Gupta, D. Berson, and J.Z. Fang, "Resource-Sensitive Profile-Directed Data Flow Analysis for Code Optimization," *The 30th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 558-568, Research Triangle Park, North Carolina, December 1997.
13. R. Gupta, D. Berson, and J.Z. Fang, "Path Profile Guided Partial Redundancy Elimination using Speculation," *IEEE International Conference on Computer Languages*, pages 230-239, Chicago, Illinois, May 1998.
14. R.N. Horspool and H.C. Ho, "Partial Redundancy Elimination Driven by a Cost-Benefit Analysis," *8th Israeli Conference on Computer Systems and Software Engineering*, pages 111-118, Herzliya, Israel, June 1997.
15. O. Ruthing, "Optimal Code Motion in Presence of Large Expressions," *IEEE International Conference on Computer Languages*, Chicago, Illinois, 1998.
16. E. Morel and C. Renvoise, "Global Optimization by Suppression of Partial Redundancies," *Communications of the ACM*, 22(2):96-103, 1979.
17. J. Knoop, O. Ruthing, and B. Steffen, "Lazy Code Motion," *Proceedings of Conference on Programming Language Design and Implementation*, pages 224-234, 1992.
18. J. Knoop, O. Ruthing, and B. Steffen, "Partial Dead Code Elimination," *Proceedings of Conference on Programming Language Design and Implementation*, pages 147-158, 1994.
19. B. Steffen, "Data Flow Analysis as Model Checking," *Proceedings TACS'91*, Sendai, Japan, Springer-Verlag, LNCS 526, pages 346-364, 1991.
20. B. Steffen, J. Knoop, and O. Rüthing, "The value flow graph: A program representation for optimal program transformations," *Proceedings of the 3rd European Symposium on Programming (ESOP'90)*, LNCS 432, pages 389–405, 1990.