We'll need some standard data flow analyses we've seen before:

$AvIn_i$ = Available In for block i

= 0 (false) for b_0

$= \text{AND}_{p \in Pred(i)} AvOut_p$

$AvOut_i = Comp_i$ OR ($AvIn_i$ AND $Transp_i$)

$\equiv Gen_i$ OR ($AvIn_i$ AND $\neg Kill_i$)

We *anticipate* an expression if it is very busy:

$AntOut_i = VeryBusyOut_i$

= 0 (false) if i is an exit block

$= \text{AND}_{s \in Succ(i)} AntIn_s$

$AntIn_i = VeryBusyIn_i$

$= AntLoc_i$ OR ($Transp_i$ AND $AntOut_i$)

Partial Availability

Partial availability is similar to available expression analysis except that an expression must be computed (and not killed) along *some* (not necessarily *all*) paths:

$PavIn_i$

= 0 (false) for b_0

$= \text{OR}_{p \in Pred(i)} PavOut_p$

$PavOut_i = Comp_i$ OR ($PavIn_i$ AND $Transp_i$)

Where are Computations Added?

The key to partial redundancy elimination is deciding where to add computations of an expression to change partial redundancies into full redundancies (which may then be optimized away).

We'll start with an "enabling term."

$Const_i = AntIn_i$ AND
$[PavIn_i$ OR $(Transp_i$ AND $\neg$ $AntLoc_i)]$

This term say that we require the expression to be:

(1) Anticipated at the start of block i (somebody wants the expression)
and
(2a) The expression must be partially available (to perhaps transform into full availability)
or
(2b) The block neither kills nor computes the expression.

Next, we compute $PPIn_i$ and $PPOut_i$. PP means "possible placement" of a computation at the start ($PPIn_i$) or end ($PPOut_i$) of a block.

These values determine whether a computation of the expression would be "useful" at the start or end of a basic block.

$PPOut_i$

= 0 (false) for all exit blocks

$= \text{AND}_{s \in Succ(i)} \; PPIn_s$

We try to move computations "up" (nearer the start block).

It makes sense to compute an expression at the end of a block if it makes sense to compute at the start of all the block's successors.

$PPIn_i$ = 0 (false) for b_0.

$= Const_i$
AND $(AntLoc_i$ OR $(Transp_i$ AND $PPOut_i))$
$\text{AND}_{p \in Pred(i)} \; (PPOut_p \text{ OR } AvOut_p)$

To determine if $PPIn_i$ is true, we first check the enabling term. It makes sense to consider a computation of the expression at the start of block i if the expression is anticipated (wanted) and partially available or if the expression is anticipated (wanted) and it is neither computed nor killed in the block.

We then check that the expression is anticipated locally or that it is unchanged within the block and possibly positioned at the end of the block.

Finally, we check that all the block's predecessors either have the expression available at their ends or are willing to position a computation at their end.

Note also, the bi-directional nature of this equation.

Inserting New Computations

After $PPIn_i$ and $PPOut_i$ are computed, we decide where computations will be inserted:

$$Insert_i = PPOut_i \text{ AND } (\neg\, AvOut_i) \text{ AND } (\neg\, PPIn_i \text{ OR } \neg\, Transp_i)$$

This rule states that we really will compute the expression at the end of block i if this is a possible placement point and the expression is not already computed and available and moving the computation still earlier doesn't work because the start of the block isn't a possible placement point or because the block kills the expression.

Removing Existing Computations

We've added computations of the expression to change partial redundancies into full redundancies. Once this is done, expressions that are fully redundant can be removed.

But where?

$$Remove_i = AntLoc_i \text{ and } PPIn_i$$

This rule states that we remove computation of the expression in blocks where it is computed locally and might be moved to the block's beginning.

Partial Redundancy Subsumes Available Expression Analysis

Using partial redundancy analysis, we can find (and remove) ordinary fully redundant available expressions.

Consider a block, b, in which:

(1) The expression is computed (anticipated) locally

and

(2) The expression is available on entrance

Point (1) tells us that $AntLoc_b$ is true

Moreover, recall that

$$PPIn_b = Const_b \text{ AND } (AntLoc_b \text{ OR } \ldots) \text{ AND } \underset{p \in Pred(i)}{\text{AND}} (AvOut_p \text{ OR } \ldots)$$

$Const_b = AntIn_b$ AND $[PavIn_b$ OR ...$]$

We know $AntLoc_b$ is true $\Rightarrow AntIn_b =$ true.

Moreover, $AvIn_b =$ true $\Rightarrow PavIn_b =$ true.

Thus $Const_b =$ true.

If $AvIn_b$ is true, $AvOut_p$ is true for all $p \in Pred(b)$.

Thus $PPIn_b$ AND $AntLoc_b$ = true = $Remove_b$

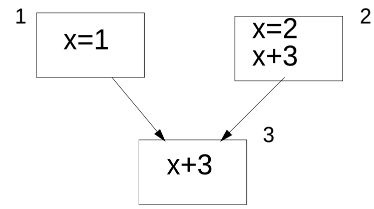
Are any computations added earlier (to any of b's ancestors)?

No:

$Insert_i = PPOut_i$ AND ($\neg$ $AvOut_i$) AND ($\neg$ $PPIn_i$ OR $\neg$ $Transp_i$)

But for any ancestor, i, between the computation of the expression and b, $AvOut_i$ is true, so $Insert_i$ must be false.

Examples of Partial Redundancy Elimination



At block 3, x+3 is partially, but not fully, redundant.

$PPIn_3$ = $Const_3$ AND ($AntLoc_3$ OR ...)

$\text{AND}_{p \in Pred(3)}$ ($PPOut_p$ OR $AvOut_p$)

$Const_3$ = $AntIn_3$ AND [$PavIn_3$ OR ...]

Now $AntIn_3$ = true and $PavIn_3$ = true.

$Const_3$ = true AND true = true

$PPout_1$ = $PPIn_3$

Default initialization of PPIn and PPOut terms is true, since we AND terms together.

$AntLoc_3$ = true.

$PPIn_3$ = true AND true

$\text{AND}_{p \in Pred(3)}$ ($PPOut_p$ OR $AvOut_p$) =

$PPOut_1$ AND $AvOut_2$= true AND true = $PPIn_3$ = $PPOut_1$.

$Insert_1$ = $PPOut_1$ AND ($\neg$ $AvOut_1$) AND ($\neg$ $PPIn_1$ OR $\neg$ $Transp_1$) =

$PPOut_1$ AND ($\neg$ $AvOut_1$) AND ($\neg$ $Transp_1$) = true,

so x+3 is inserted at the end of block 3.

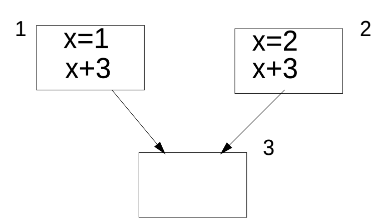
$Remove_3$ = $AntLoc_3$ and $PPIn_3$ = true AND true = true, so x+3 is removed from block 3.

Is x+3 inserted at the end of block 2? (It shouldn't be).

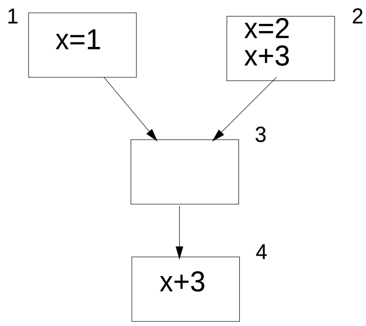
$Insert_2$ = $PPOut_2$ AND ($\neg$ $AvOut_2$) AND ($\neg$ $PPIn_2$ OR $\neg$ $Transp_2$) =

$PPOut_2$ AND false AND ($\neg$ $PPIn_2$ OR $\neg$ $Transp_2$) = false.

We now have



Computations May Move Up Several Blocks



Again, at block 4, x+3 is partially, but not fully, redundant.

$PPIn_4 = Const_4$ AND $(AntLoc_4$ OR ...)

$\underset{p \in Pred(4)}{\text{AND}} (PPOut_p \text{ OR } AvOut_p)$

$Const_4 = AntIn_4$ AND $[PavIn_4$ OR ...]
Now $AntIn_4$ = true and $PavIn_4$ = true.

$Const_4$ = true AND true = true

$PPout_3 = PPIn_4$.

$AntLoc_4$ = true.

$PPIn_4$ = true AND true

$\underset{p \in Pred(4)}{\text{AND}} (PPOut_p \text{ OR } AvOut_p) =$

$PPOut_3$ = true.

$PPIn_3 = Const_3$ AND
$((Transp_3$ AND $PPOut_3)$ OR ...)

$\underset{p \in Pred(3)}{\text{AND}} (PPOut_p \text{ OR } AvOut_p)$

$Const_3 = AntIn_3$ AND $[PavIn_3$ OR ...]
$AntIn_3$ = true and $PavIn_3$ = true.

$Const_3$ = true AND true = true

$PPOut_1 = PPIn_3$

$Transp_3$ = true.

$PPIn_3$ = true AND (true AND true)

$\underset{p \in Pred(3)}{\text{AND}} (PPOut_p \text{ OR } AvOut_p) =$

$PPOut_1$ AND $AvOut_2$= true AND true
$= PPIn_3 = PPOut_1$.

Where Do We Insert Computations?

$Insert_3 = PPOut_3$ AND $(\neg\, AvOut_3)$
AND $(\neg\, PPIn_3$ OR $\neg\, Transp_3) =$

true AND (true) AND
(false OR false) = false

so x+3 is *not* inserted at the end of block 3.

$Insert_2 = PPOut_2$ AND $(\neg\, AvOut_2)$
AND $(\neg\, PPIn_2$ OR $\neg\, Transp_2) =$

$PPOut_2$ AND (false)
AND $(\neg\, PPIn_2$ OR $\neg\, Transp_2)$=false,
so x+3 is *not* inserted at the end of block 2.

$Insert_1 = PPOut_1$ AND $(\neg\ AvOut_1)$ AND $(\neg\ PPIn_1$ OR $\neg\ Transp_1) =$

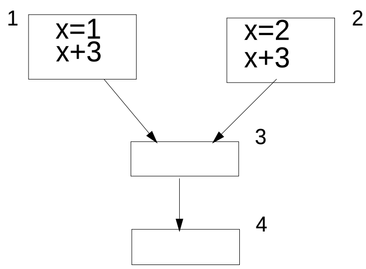
true AND (true) AND $(\neg\ PPIn_1$ OR true) = true

so x+3 *is* inserted at the end of block 3.

$Remove_4 = AntLoc_4$ and $PPIn_4$

= true AND true = true, so x+3 is removed from block 4.

We finally have



Code Movement is Never Speculative

Partial redundancy analysis has the attractive property that it never adds a computation to an execution path that doesn't use the computation.

That is, we never *speculatively* add computations.

How do we know this is so?

Assume we are about to insert a computation of an expression at the end of block b, but there is a path from b that doesn't later compute and use the expression.

Say the path goes from b to c (a successor of b), and then eventually to an end node.

Looking at the rules for insertion of an expression:

$Insert_b = PPOut_b$ AND ...

$PPOut_b = PPIn_c$ AND ...

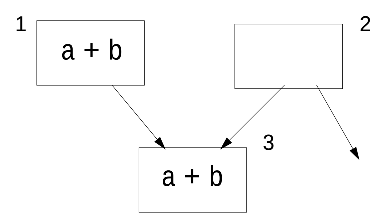
$PPIn_c = Const_c$ AND ...

$Const_c = AntIn_c$ AND ...

But if the expression isn't computed and used on the path through c, then $AntIn_c$ = False, forcing $Insert_b$ = false, a contradiction.

Can Computations Always be Moved Up?

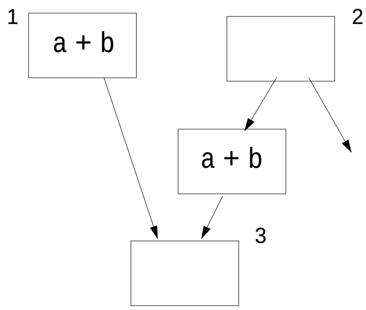
Sometimes an attempt to move a computation earlier in the CFG can be *blocked*. Consider



We'd like to move a+b into block 2, but this may be impossible if a+b isn't anticipated on all paths out of block 2.

The solution to this difficulty is no notice that we really want a+b computed on the *edge* from 2 to 3.

If we add an *artificial* block between blocks 2 and 3, movement of a+b out of block 3 is no longer blocked:



Loop Invariant Code Motion

Partial redundancy elimination subsumes loop invariant code motion.

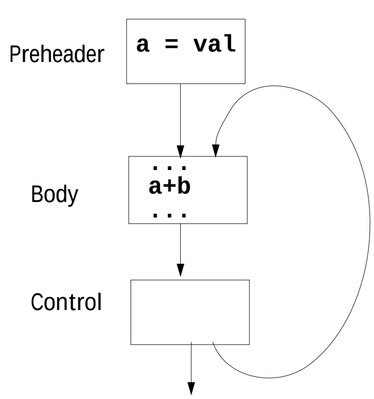
Why?

The iteration of the loop makes the invariant expression partially redundant on a path from the expression to itself.

If we're guaranteed the loop will iterate at least once (do-while or repeat-until loops), then evaluation of the expression can be anticipated in the loop's preheader.

Consider

```
a = val
do
    ...
    a+b
    ...
while (...)
```



$PPIn_B$ = $Const_B$ AND ($AntLoc_B$ OR ...) AND ($PPOut_p$ AND $AvOut_C$)

$Const_B$ = $AntIn_B$ AND [$PavIn_B$ OR ...]

$AntIn_B$ = true, $PavIn_B$ = true $\Rightarrow$ $Const_B$ = true

$PPout_P$ = $PPIn_B$, $AntLoc_B$= true, $AvOut_C$ = true $\Rightarrow$ $PPIn_B$ = true.

$Insert_P$ = $PPOut_P$ AND ($\neg$ $AvOut_P$) AND ($\neg$ $PPIn_P$ OR $\neg$ $Transp_P$) =

true AND (true) AND ($\neg$ $PPIn_P$ OR true) = true,

so we may insert a+b at the end of the preheader.

$Remove_B$ = $AntLoc_B$ and $PPIn_B$ = true AND true, so we may remove a+b from the loop body.

What About While & For Loops?

The problem here is that the loop may iterate zero times, so the loop invariant isn't really very busy (anticipated) in the preheader.

We can, however, change a while (or for) into a do while:

```
while (expr){    if (expr)
   body       ≡    do {body}     ≈
}                  while (expr)


goto L:
  do {body}
 L:
  while (expr)
```

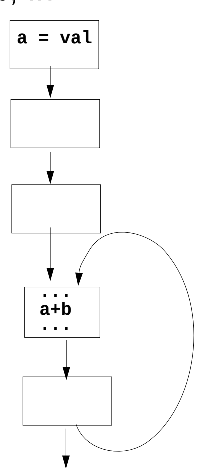
After we know the loop will iterate once, we can evaluate the loop invariant.

Code Placement in Partial Redundancy Elimination

While partial redundancy elimination correctly places code to avoid unnecessary reevaluation of expressions along execution paths, its choice of code placement can sometimes be disappointing.

It always moves an expression back as far as possible, as long as computations aren't added to unwanted execution paths. This may unnecessarily lengthen live ranges, making register allocation more difficult.

For example, in



where will we insert a+b?

$\text{Insert}_P = \text{PPOut}_P \text{ AND } (\neg \text{ AvOut}_P)$
$\quad \text{AND } (\neg \text{ PPIn}_P \text{ OR } \neg \text{ Transp}_P)$

The last term will be true at the top block, but not elsewhere.

In "Lazy Code Motion" (PLDI 1992), Knoop, Ruething and Steffan show how to eliminate partial redundancies while minimizing register pressure.

Their technique seeks to evaluate an expression as "late as possible" while still maintaining computational optimality (no redundant or unnecessary evaluations on *any* execution paths).

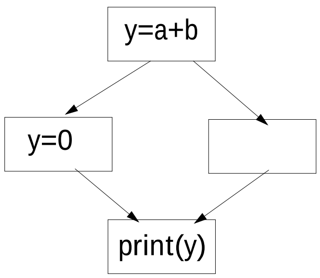
Their technique places loop invariants in the loop preheader rather than in an earlier predecessor block as Morel & Renvoise do.

Partial Dead Code Elimination

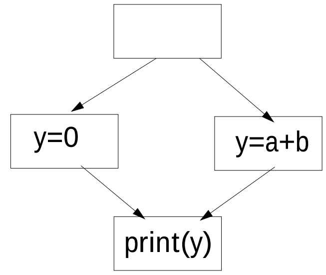
Partial Redundancy Elimination aims to never reevaluate an expression on any path, and never to add an expression on any path where it isn't needed.

These ideas suggest an interesting related optimization—eliminating expressions that are partially dead.

Consider



On the left execution path, a+b is dead, and hence useless. We'd prefer to compute a+b only on paths where it is used, obtaining



This optimization is investigated in "Partial Dead Code Elimination" (PLDI 1994), Knoop, Ruething and Steffan.

This optimization "sinks" computations onto paths where they are needed.