SSA and Value Numbering

We already know how to do available expression analysis to determine if a previous computation of an expression can be reused.

A limitation of this analysis is that it can't recognize that two expressions that aren't syntactically identical may actually still be equivalent.

For example, given

```
t1 = a + b
c = a
t2 = c + b
```

Available expression analysis won't recognize that **t1** and **t2** must be equivalent, since it doesn't track the fact that **a** = **c** at **t2**.

Value Numbering

An early expression analysis technique called *value numbering* worked only at the level of basic blocks. The analysis was in terms of "values" rather than variable or temporary names.

Each non-trivial (non-copy) computation is given a number, called its *value number*.

Two expressions, using the same operators and operands with the same value numbers, must be equivalent.

For example,

```
t1 = a + b
c = a
t2 = c + b
```

is analyzed as

```
v1 = a
v2 = b
t1 = v1 + v2
c = v1
t2 = v1 + v2
```

Clearly **t2** is equivalent to **t1** (and hence need not be computed).

In contrast, given

```
t1 = a + b
a = 2
t2 = a + b
```

the analysis creates

```
v1 = a
v2 = b
t1 = v1 + v2
v3 = 2
t2 = v3 + v2
```

Clearly **t2** is not equivalent to **t1** (and hence will need to be recomputed).

Extending Value Numbering to Entire CFGs

The problem with a global version of value numbering is how to reconcile values produced on different flow paths. But this is exactly what SSA is designed to do!

In particular, we know that an ordinary assignment

`x = y`

does *not* imply that all references to x can be replaced by y after the assignment. That is, an assignment *is not* an assertion of value equivalence.

But,

in SSA form

$x_i = y_j$

does mean the two values are *always* equivalent after the assignment. If y_j reaches a use of x_i, that use of x_i *can* be replaced with y_j.

Thus in SSA form, an assignment *is* an assertion of value equivalence.

We will assume that simple variable to variable copies are removed by substituting equivalent SSA names.

This alone is enough to recognize some simple value equivalences.

As we saw,

$t_1 = a_1 + b_1$

$c_1 = a_1$

$t_2 = c_1 + b_1$

becomes

$t_1 = a_1 + b_1$

$t_2 = a_1 + b_1$

Partitioning SSA Variables

Initially, all SSA variables will be partitioned by the *form* of the expression assigned to them.

Expressions involving different constants or operators won't (in general) be equivalent, even if their operands happen to be equivalent.

Thus

$v_1 = 2$ and $w_1 = a_2 + 1$

are always considered inequivalent.

But,

$v_3 = a_1 + b_2$ and $w_1 = d_1 + e_2$

may *possibly* be equivalent since both involve the same operator.

Phi functions are potentially equivalent only if they are in the same basic block.

All variables are initially considered equivalent (since they all initially are considered uninitialized until explicit initialization).

After SSA variables are grouped by assignment form, groups are split.

If a_i op b_y and c_k op d_l are in the same group (because they both have the same operator, op) and $a_i \not\equiv c_k$ or $b_j \not\equiv d_l$ then we split the two expressions apart into different groups.

We continue splitting based on operand inequivalence, until no more splits are possible. Values still grouped are equivalent.

Example

```
if (...) {
  a1=0
  if (...)
    b1=0
  else {
    a2=x0
    b2=x0 }
  a3=φ(a1,a2)
  b3=φ(b1,b2)
  c2=*a3
  d2=*b3 }
else {
  b4=10 }
a5=φ(a0,a3)
b5=φ(b3,b4)
c3=*a5
d3=*b5
e3=*a5
```

Initial Groupings:

$G_1=[a_0,b_0,c_0,d_0,e_0,x_0]$
$G_2=[a_1=0,\ b_1=0]$
$G_3=[a_2=x_0,\ b_2=x_0]$
$G_4=[b_4=10]$
$G_5=[a_3=\phi(a_1,a_2),\ b_3=\phi(b_1,b_2)]$
$G_6=[a_5=\phi(a_0,a_3),\ b_5=\phi(b_3,b_4)]$
$G_7=[c_2=*a_3,\ d_2=*b_3,\ d_3=*b_5,\ c_3=*a_5,\ e_3=*a_5]$

Now b_4 isn't equivalent to anything, so split a_5 and b_5. In G_7 split operands b_3, a_5 and b_5. We now have


```
if (...) {
  a1=0
  if (...)
    b1=0
  else {
    a2=x0
    b2=x0 }
  a3=φ(a1,a2)
  b3=φ(b1,b2)
  c2=*a3
  d2=*b3 }
else {
  b4=10 }
a5=φ(a0,a3)
b5=φ(b3,b4)
c3=*a5
d3=*b5
e3=*a5
```

Final Groupings:

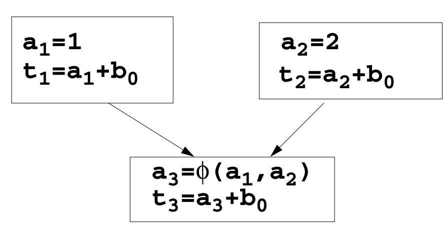
$G_1=[a_0,b_0,c_0,d_0,e_0,x_0]$
$G_2=[a_1=0,\ b_1=0]$
$G_3=[a_2=x_0,\ b_2=x_0]$
$G_4=[b_4=10]$
$G_5=[a_3=\phi(a_1,a_2),\ b_3=\phi(b_1,b_2)]$
$G_{6a}=[a_5=\phi(a_0,a_3)]$
$G_{6b}=[b_5=\phi(b_3,b_4)]$
$G_{7a}=[c_2=*a_3,\ d_2=*b_3]$
$G_{7b}=[d_3=*b_5]$
$G_{7c}=[c_3=*a_5,\ e_3=*a_5]$

Variable e_3 can use c_3's value and d_2 can use c_2's value.

Limitations of Global Value Numbering

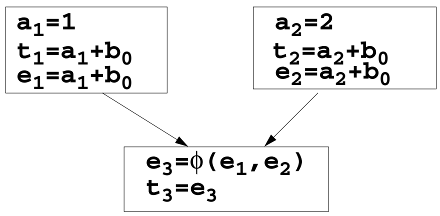
As presented, our global value numbering technique doesn't recognize (or handle) computations of the same expression that produce different values along different paths.

Thus in



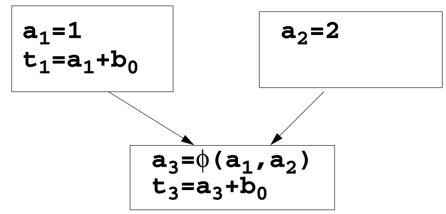
variable a_3 isn't equivalent to either a_1 or a_2.

But,
we can still remove a redundant computation of **a+b** by moving the computation of t_3 to each of its predecessors:

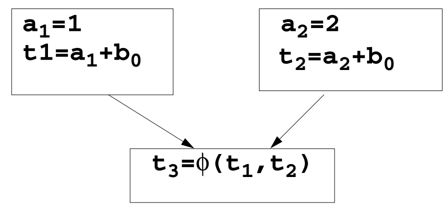


Now a redundant computation of **a+b** is evident in each predecessor block. Note too that this has a nice register targeting effect—e_1, e_2 and e_3 can be readily mapped to the same live range.

The notion of moving expression computations above phi functions also meshes nicely with notion of partial redundancy elimination. Given



moving **a+b** above the phi produces



Now **a+b** is computed only once on each path, an improvement.

Reading Assignment

- Read "Global Optimization by Suppression of Partial Redundancies," Morel and Renvoise.
  (Linked from the class Web page.)
- Read "Profile Guided Code Positioning," Pettis and Hansen.
  (Linked from the class Web page.)

Partial Redundancy Analysis

Partial Redundancy Analysis is a boolean-valued data flow analysis that generalizes available expression analysis.

Ordinary available expression analysis tells us if an expression must already have been evaluated (and not killed) along *all* execution paths.

Partial redundancy analysis, originally developed by Morel & Renvoise, determines if an expression has been computed along *some* paths. Moreover, it tells us where to add new computations of the expression to change a partial redundancy into a full redundancy.

This technique *never* adds computations to paths where the computation isn't needed. It strives to avoid having any redundant computation on any path.

In fact, this approach includes movement of a loop invariant expression into a preheader. This loop invariant code movement is just a special case of partial redundancy elimination.

Basic Definition & Notation

For a Basic Block i and a particular expression, e:

$Transp_i$ is true if and only if e's operands aren't assigned to in i.

$Transp_i \equiv \neg Kill_i$

$Comp_i$ is true if and only if e is computed in block i and is not killed in the block after computation.

$Comp_i \equiv Gen_i$

$AntLoc_i$ (Anticipated Locally in i) is true if and only if e is computed in i and there are no assignments to e's operands prior to e's computation.

If $AntLoc_i$ is true, computation of e in block i will be redundant if e is available on entrance to i.

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

$AvIn_i$ = Available In for block i

$= 0$ (false) for b_0

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

$$AvOut_i = Comp_i \text{ OR } (AvIn_i \text{ AND } Transp_i)$$

$$\equiv Gen_i \text{ OR } (AvIn_i \text{ AND } \neg Kill_i)$$

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

$AntOut_i = VeryBusyOut_i$

$= 0$ (false) if i is an exit block

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

$AntIn_i = VeryBusyIn_i$

$= AntLoc_i$ 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

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

$PavOut_i = Comp_i$ 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

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

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.

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

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?

$Remove_i = AntLoc_i$ 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

$PPIn_b$ = $Const_b$ AND
($AntLoc_b$ OR ...)

AND$_{p \in Pred(i)}$ ($AvOut_p$ OR ...)

$Const_b$ = $AntIn_b$ 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 = $Remove_b$

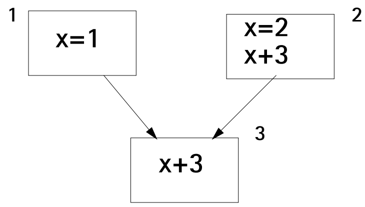
Are any computations added earlier (to any of b's ancestors)?

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 ...)

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

$Const_3$ = $AntIn_3$ 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

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

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

$Insert_1$ = $PPOut_1$ 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.

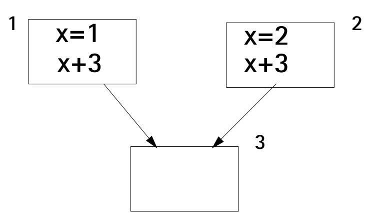
$Remove_3$ = $AntLoc_3$ and $PPIn_3$ = true AND true = true, so x+3 is removed from block 3.

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

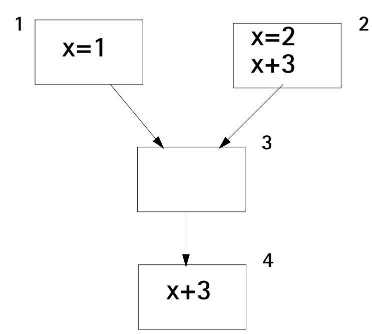
$Insert_2$ = $PPOut_2$ AND ($\neg$ $AvOut_2$) AND ($\neg$ $PPIn_2$ OR $\neg$ $Transp_2$) =

$PPOut_2$ 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

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

$PPOut_3$ = true.

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

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

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