## Redundancy Elimination

**Aim:** Eliminate redundant operations in dynamic execution

**Why occur?**
- *Loop-invariant code:* Ex: constant assignment in loop
- *Same expression computed:* Ex: addressing

Value numbering is an example

Requires dataflow analysis

Other optimizations:
- *Constant subexpression elimination*
- *Loop-invariant code motion*
- *Partial redundancy elimination*

---

## Common Subexpression Elimination

Replace recomputation of expression by use of temp which holds value

- Ex. (s1) \( y := a + b \) \( \rightarrow \) Ex. (s1) \( \text{temp} := a + b \)
  
  (s1') \( y := \text{temp} \)

(s2) \( z := a + b \) \( \rightarrow \) (s2) \( z := \text{temp} \)

**Illegal?**

**How different from value numbering?**

- Ex. (s1) \( \text{read(i)} \)
  
  (s2) \( j := i + 1 \)
  
  (s3) \( k := i \)
  
  (s4) \( l := k + 1 \)

**Why need temp?**

Local and Global
Local CSE (BB)

Ex. (s1)  \( c := a + b \)  
(s2)  \( d := m \& n \)  
(s3)  \( e := a + b \)  
(s4)  \( m := 5 \)  
(s5)  \( \text{if}(m \& n) \ldots \)  

\( (s1) \ t_1 := a + b \)  
(s2)  \( d := m \& n \)  
(s3)  \( e := t_1 \)  
(s4)  \( m := 5 \)  
(s5)  \( \text{if}(m \& n) \ldots \)  

5 instr, 4 ops, 7 vars  
6 instr, 3 ops, 8 vars  

Always better?  
Method: keep track of expressions computed in block whose operands have not changed value

CSE Hash Table

\[
\begin{array}{c|c}
(\+, a, b) & (s1) \ t_1 := a + b \\
(\&, m, n) & \\
\end{array}
\]

Global CSE example

Assumes \( b \) is used later
Global CSE

An expression e is available at entry to B if on every path p from Entry to B, there is an evaluation of e at B’ on p whose values are not redefined between B’ and B.

Solve by:
1. Find Available Expressions (Data flow problem)
2. For each available expression e

\[
\begin{align*}
x+y & \quad x+y & \quad x+y \\
\downarrow & & \downarrow \\
x+y & \quad t & \quad t \\
\end{align*}
\]

Do backward search from e in CFG to find the evaluations of e
Create temp t to hold previous evaluations, and replace e by t

Solving Available Expressions

**Gen(B)** set of expressions evaluated in B available on exit from B

**Kill(B)** set of expressions killed by B

\begin{align*}
c := a + b & \quad Gen(B) = \{w\&z\} \\
d := w \& z & \quad Kill(B) = \{\text{all expr. involving } c,d,a,e\} \\
a := 5 & \\
e := e - 7 &
\end{align*}

Forward or Backward?

\[
AEin(B) = \bigcap p \text{ pred of } B \text{ AEout}(p)
\]

\[
AEout(B) = Gen(B) \bigcup (AEin(B) - Kill(B))
\]
Applying CSE

```
  a + b  a + b  a + b  t1 := a + b  t1 := a + b  t2 := a + b
    a + b   a + b   t1         t1
      t2
```

Create new temps for each occurrence of expression--necessary?

Is CSE always desirable?

When should CSE be applied?

**Forward Substitution: Inverse of CSE**

Replace copy by reevaluation of expression

Legal?

Loop-invariant Code Motion

Move computations that do not vary in loop outside of loop (pre-header)

**Ex.**
```
do i := 1, 100
  l := i + (c * 5)
do j := 1, 100
  a(i,j) := 100*c  + 10*i +j
door
endo
endo
t1 := (100*c)
t2 := (c * 5)
do i := 1, 100
  l := i + t2
do j := 1, 100
  a(i,j) := t3 +j
door
endo
endo
t1 := a + b  t1 := a + b  t2 := a + b
    t1
```

Almost always a good idea

Especially useful on addressing code

Need def-use information (dataflow analysis)
Finding Code that is Loop-invariant

Find loops

From inner to outer loop order, repeat till no change:

Mark all constant operands as loop-invar.

Mark all operands with all its defs from outside loop-inv.

Mark all expressions whose operands are loop-inv. as loop-inv.

Mark all instructions whose only defs are loop-inv. as loop-inv.

Problems
Safe to move loop invariant code?

Assignments: NO

If move executed ass’t to preheader:

   May change data flow
   May raise exceptions that would not o.w. be raised

Conditional tests: YES, if

   No side effects (ie no assignments)
   Changes in C.D. are allowed

May be redundantly executed....

Code motion of assignments

1. Don’t move asst unless
   asst. dominates all uses, and
   asst dominates all exits of loop
2. Move with conditional (if possible)
Problems with Code Motion

Procedure Calls

\[ y := \text{rand()} \]

Value may be different each call, even if arguments the same

Fix:

1. Don’t move, or
2. Interprocedural analysis

Partial Redundancy Elimination (PRE)

An expression is **partially redundant** at node \( n \) if it is evaluated on some path from the entry to \( n \), and there are no redefinitions of any of its operands on the path after the last evaluation.

Elimination

*Discovers partial redundancies*

*Converts to full redundancies*

*Removes redundancy*
Code Hoisting

Application of Very Busy Expressions

Def. An expression \( e \) is very busy at \( B \) if on all paths from \( B \) to exit, 
\( e \) is evaluated before any of its operands are defined.

B is potential point to hoist code, may not be legal!
Want earliest such point

Beneficial?

May not always improve execution time

Example
Loop Unrolling

Copy body of loop k times, substituting successive indices and adjusting outer loop accordingly

Ex. \( \text{do } i = 1,99 \)
\[ a(i) := b(i) + c \]
\[ \text{endo} \]

Ex. \( \text{do } i = 1,99,3 \)
\[ a(i) := b(i) + c \]
\[ a(i+1) := b(i+1) + c \]
\[ a(i+2) := b(i+2) + c \]
\[ \text{endo} \]

Always possible?

Beneficial?

Loop Reversal

Run the loop backwards

Ex. \( \text{for } i=1,n \)
\[ a[i] := b[i] + 1 \]
\[ c[i] := a[i] - 2 \]
\[ \text{endfor} \]
\[ \text{for } i=1,n \]
\[ d[i] := 1/c[i+1] \]
\[ \text{endfor} \]

Ex. \( \text{for } i=n,1 \)
\[ a[i] := b[i] + 1 \]
\[ c[i] := a[i] - 2 \]
\[ \text{endfor} \]
\[ \text{for } i=n,1 \]
\[ d[i] := 1/c[i+1] \]
\[ \text{endfor} \]

Can't fuse two loops

Can fuse two loops

Illegal if loop-carried dependences

Reverses dependence direction
Loop Interchange

Interchange

Ex. for $i=1,n$
for $j=1,n$

for $j=1,n$
for $i=1,n$

Not always legal

Can get any permutation of loops by composing interchanges

Can be used to implement tiling

Tiling

Partitioning of ISG into regular size and shape pieces, + schedule for executing each piece

Can be done by strip mine + interchange

Ex. do $i = 1,n$
do $j = 1,n$

a[$i,j$] := a[$i,j$] + b[$i,j$]
c[$i,j$] := a[$i-1,j-1$]*2

Do ss by ss block at a time