

## 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*

1

## Common Subexpression Elimination

Replace recomputation of expression by use of temp which holds value

Ex. (s1)  $y := a + b$

→

Ex. (s1)  $temp := a + b$   
(s1')  $y := temp$

(s2)  $z := a + b$

(s2)  $z := temp$

Illegal?

How different from value numbering?

Ex. (s1)  $read(i)$

(s2)  $j := i + 1$

(s3)  $k := i$

(s4)  $l := k + 1$

$i + 1, k + 1$

*no cse, same value number*

Why need temp?

Local and Global

2

## Local CSE (BB)

<p>Ex. (s1) <math>c := a + b</math>          (s2) <math>d := m \&amp; n</math>          (s3) <math>e := a + b</math>          (s4) <math>m := 5</math>          (s5) <math>\text{if}(m \&amp; n) \dots</math></p>	$\longrightarrow$	<p>(s1) <math>t1 := a + b</math>          (s1') <math>c := t1</math>          (s2) <math>d := m \&amp; n</math>          (s3) <math>e := t1</math>          (s4) <math>m := 5</math>          (s5) <math>\text{if}(m \&amp; n) \dots</math></p>
---	-------------------	---

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

Always better?

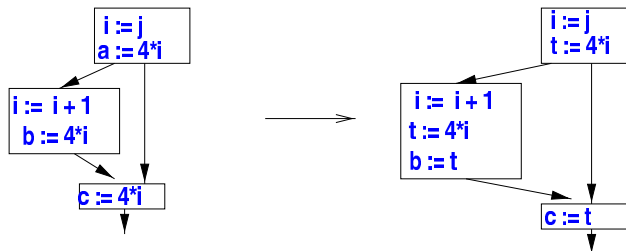
Method: keep track of expressions computed in block whose operands have not changed value

### CSE Hash Table

(+, a, b)
(&, m, n)

3

## Global CSE example



*Assumes b is used later*

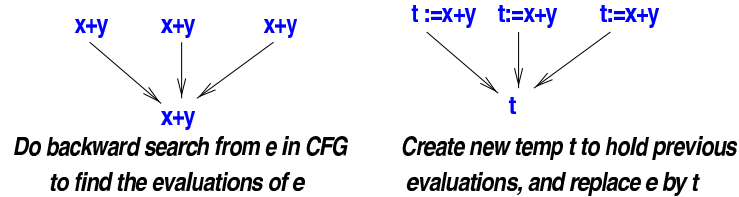
4

## 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  $e$



5

## Solving Available Expressions

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

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

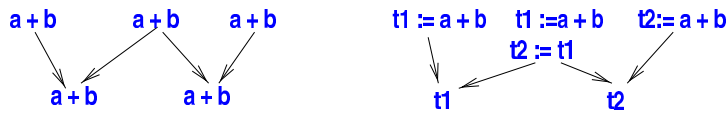
$c := a + b$        $Gen(B) = \{w \& z\}$   
 $d := w \& z$        $Kill(B) = \{all\ expr.\ involving\ c, d, a, e\}$   
 $a := 5$   
 $e := e - 7$

Forward or Backward?

$AEin(B) = \bigcap_{p \text{ pred of } B} AEout(p)$   
 $AEout(B) = Gen(B) \cup (AEin(B) - Kill(B))$

6

## Applying CSE



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?

7

## 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
    enddo
enddo

t1 := (100*c)
t2 := (c * 5)
do i := 1, 100
    l := i + t2
    t3 := t1 + 10 * i
    do j := 1, 100
        a(i,j) := t3 + j
    enddo
enddo
    
```

Almost always a good idea

Especially useful on addressing code

Need def-use information (dataflow analysis)

8

# 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.*

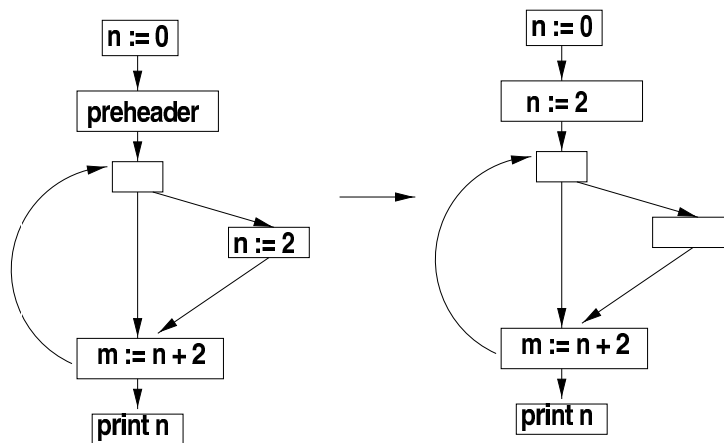
*Mark all operands with all its defs from outside loop-inv.*

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

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

9

## Problems



10

## 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)

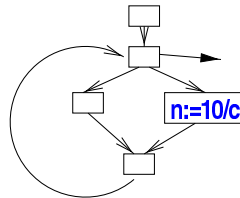
Changes in C.D. are allowed

*May be redundantly executed....*

11

## 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)*

12

## Problems with Code Motion

Procedure Calls

```
y:= rand()
```

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

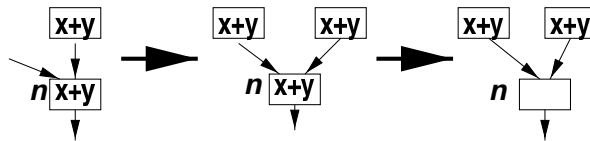
Fix:

1. *Don't move, or*
2. *Interprocedural analysis*

13

## 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**

14

# 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!

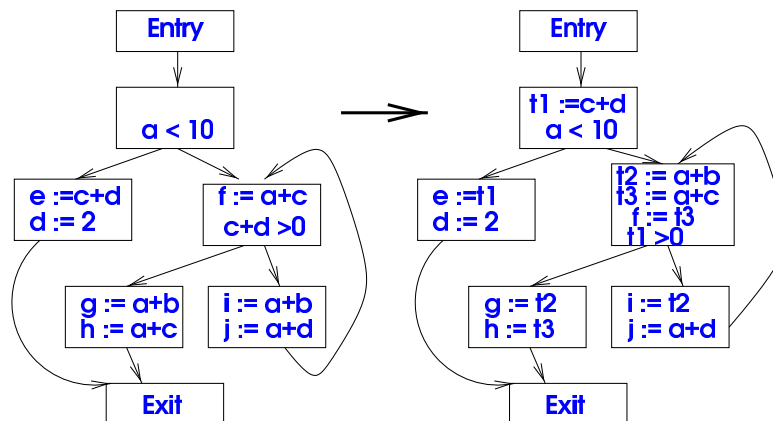
Want earliest such point

**Beneficial?**

*May not always improve execution time*

15

## Example



16

## Induction Variables & Strength Reduction

Induction variable: *variable whose successive values form an arithmetic progression in loop*

```
Ex. do i = 1,99
    a(i) := 2*i-1
enddo
```

*i is inductive*  
*2\*i-1 is inductive*

$\Rightarrow$

```
t := -1
do i = 1,99
    t := t+2
    a(i) := t
enddo
```

Strength Reduction

*Replaces expensive ops (like \*) with less expensive ops (like +)*

17

## Loop Peeling

Remove 1 or more iterations from beginning/end of loop

```
Ex. do i = 1,100
    a(i) := a(i) + a(1)
enddo
```

*i = 1*

$\Rightarrow$

```
a(1) := a(1) + a(1)
do i = 2,100
    a(i) := a(i) + a(1)
enddo
```

*i > 1*

*Beneficial?*

*Always possible?*

18

## Index Set Splitting

---

Split index set 1,...,N into multiple sets

```
Ex. do i = 1,100
    a(i) := b(i) + c(i)
    if (i>50) then
        d(i) := a(i) + a(i-10)
    enddo
                                =>
do i = 1,50
    a(i) := b(i) + c(i)
enddo
do i = ,51,100
    a(i) := b(i) + c(i)
    d(i) := a(i) + a(i-10)
enddo
```

*Beneficial?*

*Always possible?*

19

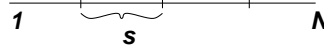
## Strip Mining

---

Turn 1 loop into 2:

*Outer: by strip size s*

*Inner: inside each strip*



```
Ex. do i = 1,100
    a(i+20) := a(i) + b
enddo
                                =>
do is = 1,100,20
    do i = is, is+19
        a(i+20) := a(i) + b
    enddo
enddo
```

*Always possible?*

*Beneficial?*

20

## Loop Unrolling

---

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

```
Ex. do i = 1,99
    a(i) := b(i) + c
enddo
```

→

```
. do i = 1,99,3
    a(i) := b(i) + c
    a(i+1) := b(i+1) + c
    a(i+2) := b(i+2) + c
enddo
```

*Always possible?*

*Beneficial?*

21

## Loop Reversal

---

Run the loop backwards

```
Ex. for i=1,n
    a[i] := b[i] + 1
    c[i] := a[i] - 2
endfor
for i=1,n
    d[i] := 1/c[i+1]
endfor
```

Can't fuse two loops

```
Ex. for i= n,1
    a[i] := b[i] + 1
    c[i] := a[i] - 2
endfor
for i= n,1
    d[i] := 1/c[i+1]
endfor
```

Can fuse two loops

*Illegal if loop-carried dependences*

*Reverses dependence direction*

22



# Unimodular Transformations

Any number of interchanges, skews, reversals

Can be represented by square integer matrices with  $\det. = +1$

*Interchange*  $i,j \longrightarrow j,i$   $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$

*Reversal*  $i,j \longrightarrow i,-j$   $\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$

*Skew*  $i,j \longrightarrow i, i+j$   $\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$

Easy test for legality, using matrix representation

25

# 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
    enddo
enddo
    
```

$\xrightarrow{\text{sm}}$

```

do it = 1, n, ss
    do i = it, min(n, it+ss-1)
        do jt = 1, n, ss
            do j = jt, min(n, jt+ss-1)
                a[i,j] := ...
                c[i,j] := ...
            enddo
        enddo
    enddo
enddo
    
```

$\sum i$

Do ss by ss block at a time

26