Reuse Optimization

Last time

- Common subexpression elimination (CSE)

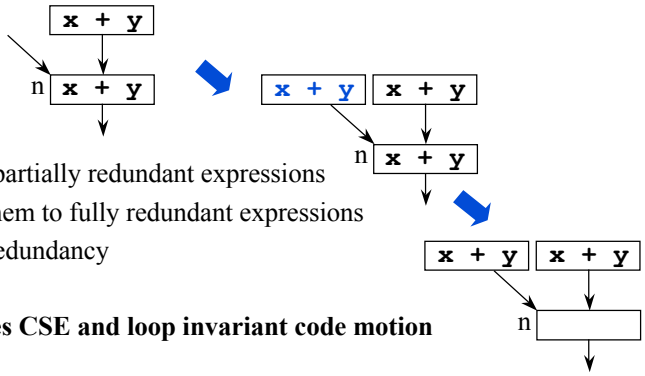
Today

- Partial redundancy elimination (PRE)

Partial Redundancy Elimination (PRE)

Partial Redundancy

- An expression (*e.g.,* **`x+y`**) is **partially redundant** at node n if **some** path from the entry node to n evaluates **`x+y`**, and there are no definitions of **`x`** or **`y`** between the last evaluation of **`x+y`** and n



Elimination

- Discover partially redundant expressions
- Convert them to fully redundant expressions
- Remove redundancy

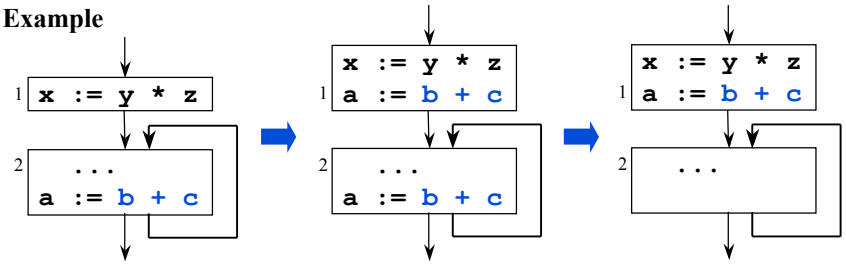
PRE subsumes CSE and loop invariant code motion

Loop Invariance Example

PRE removes loop invariants

- An invariant expression is partially redundant
- PRE converts this partial redundancy to full redundancy
- PRE removes the redundancy

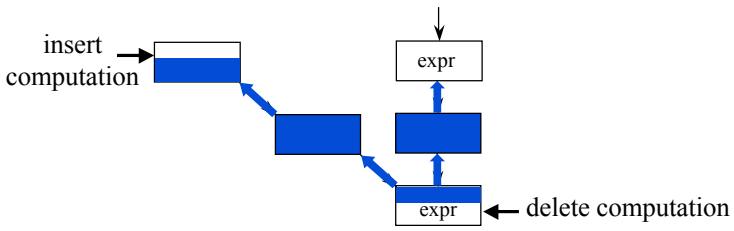
Example



Implementing PRE [Morel & Renvoise 1979]

Big picture

- Use local properties (**availability** and **anticipability**) to determine where redundancy can be created within a basic block
- Use global analysis (data-flow analysis) to discover where partial redundancy can be converted to full redundancy
- Insert code and remove redundant expressions



Local Properties

An expression is locally **transparent** in block b if its operands are not modified in b

An expression is locally **available** in block b if it is computed at least once and its operands are not modified after its last computation in b

An expression is locally **anticipated** if it is computed at least once and its operands are not modified before its first evaluation

Example

```
a := b + c
d := a + e
```

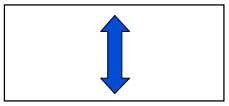
Transparent: {`b + c`}
Available: {`b + c, a + e`}
Anticipated: {`b + c`}

Local Properties (cont)

How are these properties useful?

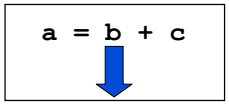
– They tell us where we can introduce redundancy

Transparent



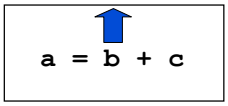
The expression can be redundantly evaluated anywhere in the block

Available



The expression can be redundantly evaluated anywhere after its last evaluation in the block

Anticipated

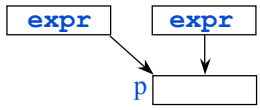


The expression can be redundantly evaluated anywhere before its first evaluation in the block

Global Availability

Intuition

- Global availability is the same as Available Expressions
- If e is globally available at p, then an evaluation at p will create redundancy along all paths leading to p



Flow Functions

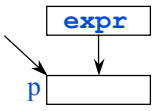
available_in[n] = $\bigcap_{p \in pred[n]}$ available_out[p]

available_out[n] = locally_available[n] $\cup$
(available_in[n] $\cap$ transparent[n])

(Global) Partial Availability

Intuition

- An expression is partially available if it is available along some path
- If e is partially available at p, then ∃ a path from the entry node to p such that the evaluation of e at p would give the same result as the previous evaluation of e along the path



Flow Functions

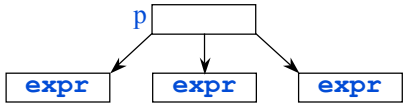
partially_available_in[n] = $\bigcup_{p \in pred[n]}$ partially_available_out[p]

partially_available_out[n] = locally_available[n] $\cup$
(partially_available_in[n] $\cap$ transparent[n])

Global Anticipability

Intuition

– If e is globally anticipated at p, then an evaluation of e at p will make the next evaluation of e redundant along all paths from p



Flow Functions

$$\text{anticipated\_out}[n] = \bigcap_{s \in succ[n]} \text{anticipated\_in}[s]$$

$$\text{anticipated\_in}[n] = \text{locally\_anticipated}[n] \cup (\text{anticipated\_out}[n] \cap \text{transparent}[n])$$

Global Possible Placement

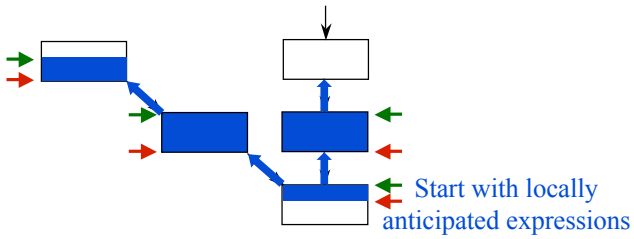
Goal

– Convert partial redundancies to full redundancies

– Possible Placement uses a backwards analysis to identify locations where such conversions can take place

– e ∈ ppin[n] can be placed at entry of n

– e ∈ ppout[n] can be placed at exit of n



Push Possible Placement backwards as far as possible

Global Possible Placement (cont)

The placement will create a redundancy on every edge out of the block

Flow Functions

Will turn partial redundancy into full redundancy

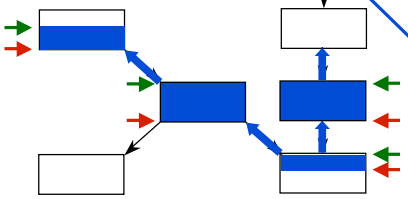
ppout[n] = $\bigcap_{s \in succ[n]}$ ppin[s]

ppin[n] = anticipated_in[n] $\bigcap$ partially_available_in[n] $\bigcap$

(locally_anticipated[n] $\bigcup$ (ppout[n] $\bigcap$ transparent[n]))

Middle of chain

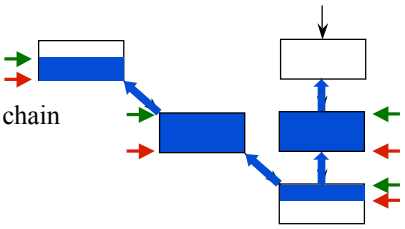
This block is at the beginning of a chain



Updating Blocks

Intuition

- Perform insertions at top of the chain
- Perform deletion at the bottom of the chain



Functions

- delete[n] = ppin[n] $\bigcap$ locally_anticipated[n]

- insert[n] = ppout[n]

  $\bigcap$ (¬ppin[n] $\bigcup$ ¬ transparent[n])

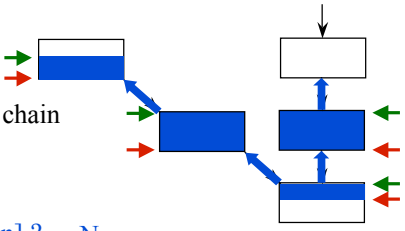
  $\bigcap$ ¬available_out[n]

Don't insert it where it's fully redundant

Updating Blocks (cont)

Intuition

- Perform insertions at top of the chain
- Perform deletion at the bottom of the chain



Functions

¬ ppout[n] ? No

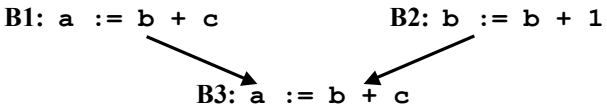
- delete[n] = ppin[n] ∩ ~~locally_anticipated[n]~~

- insert[n] = ppout[n]
  ∩ (¬ppin[n] ∪ ~~¬ transparent[n]~~) Can we omit this clause?
  ∩ ¬available_out[n]

Sandwich Example

	B0	B1	B2	B3	B4
transparent		a+b		a+b	a+b
locally_available		a+b	a+b	a+b	a+b
locally_anticipated	a+b	a+b	a+b	a+b	a+b
available_in					a+b
available_out		a+b	a+b	a+b	a+b
partially_available_in	a+b		a+b		a+b
partially_available_out		a+b	a+b	a+b	a+b
anticipated_out	a+b	a+b	a+b	a+b	
anticipated_in	a+b	a+b	a+b	a+b	a+b
ppout	a+b	a+b	a+b	a+b	
ppin	a+b		a+b		a+b
insert	a+b				
delete			a+b		a+b

Example



	B1	B2	B3
transparent	{b+c}		{b+c}
locally_available	{b+c}		{b+c}
locally_anticipated	{b+c}	{b+1}	{b+c}
available_in			
available_out	{b+c}		{b+c}
partially_available_in			{b+c}
partially_available_out	{b+c}		{b+c}
anticipated_out	{b+c}	{b+c}	
anticipated_in	{b+c}	{b+1}	{b+c}
ppout	{b+c}	{b+c}	
ppin			{b+c}
insert		{b+c}	
delete			{b+c}

Comparing Redundancy Elimination

Value numbering
- Examines values not expressions
- Symbolic

CSE
- Examines expressions

PRE
- Examines expressions
- Subsumes CSE and loop invariant code motion
- Other implementations are now available

Constant propagation
- Requires that values be statically known

PRE Summary

What's so great about PRE?

- A modern optimization that subsumes earlier ideas
- Composes several simple data-flow analyses to produce a powerful result
  - Finds earliest and latest points in the CFG at which an expression is anticipated

Next Time

Assignments

- HW2 has been posted, start it now!

Lecture

- Alias analysis