Loop Invariant Code Motion

Last Time

- Loop invariant code motion
- Value numbering

Today

- Finish value numbering
- More reuse optimization
  - Common subexpression elimination
  - Partial redundancy elimination

Next Time

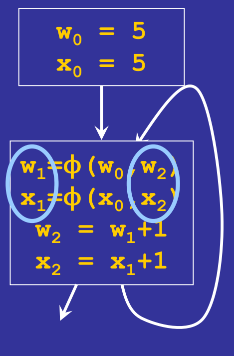
- Something special

February 25, 2015 Reuse Optimization II 1

Comparing Optimistic and Pessimistic Approaches

Differences

- Handling of loops
- Pessimistic makes worst-case assumptions on back edges
- Optimistic requires actual contradiction to split classes



February 25, 2015 Reuse Optimization II 2

Role of SSA

Single global result

– Variables correspond to values

a not congruent to anything

```
a = b
.  .  .
a = c
.  .  .
a = d
```

Congruence classes: $\{a_1,b\}$, $\{a_2,c\}$, $\{a_3,d\}$

a_1 = b
. . .
a_2 = c
. . .
a_3 = d

No data flow analysis

– Optimistic: Iterate over congruence classes, not CFG nodes
– Pessimistic: Visit each assignment once

ϕ-functions

– Make data-flow merging explicit
– Treat like normal functions

Reuse Optimization

More reuse optimization

– Common subexpression elimination (CSE)
– Partial redundancy elimination (PRE)

Common Subexpression Elimination

Idea

- Find common subexpressions whose range spans the same basic blocks and eliminates unnecessary re-evaluations
- Leverage available expressions

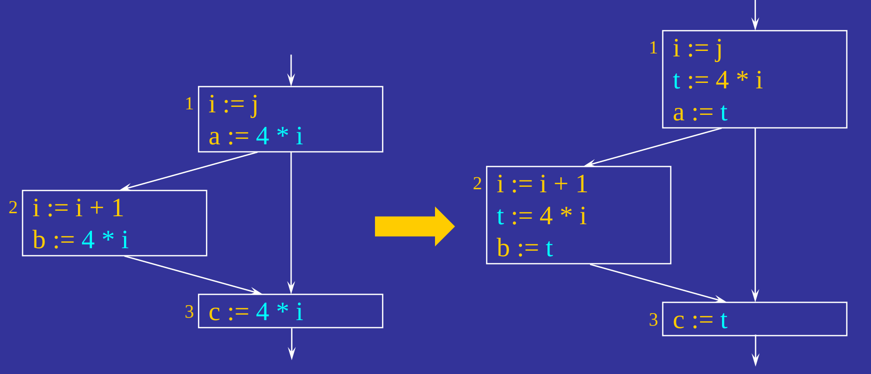
Recall available expressions

- An expression (*e.g.*, **`x+y`**) is **available** at node n if every path from the entry node to n evaluates **`x+y`**, and there are no definitions of **`x`** or **`y`** after the last evaluation along that path

Strategy

- If an expression is available at a point where it is evaluated, it need not be recomputed

CSE Example



Will value numbering find this redundancy?

- No; value numbering operates on values
- CSE operates on expressions

Is CSE strictly better than value numbering?

Another CSE Example

Before CSE

```
c := a + b
d := m & n
e := b + d
f := a + b
g := -b
h := b + a
a := j + a
k := m & n
j := b + d
a := -b
if m & n goto L2
```

Summary
11 instructions
12 variables
9 binary operators

After CSE

```
t1 := a + b
c := t1
t2 := m & n
d := t2
t3 := b + d
e := t3
f := t1
g := -b
h := t1
a := j + a
k := t2
j := t3
a := -b
if t2 goto L2
```

Summary
14 instructions
15 variables
4 binary operators

Which is better?

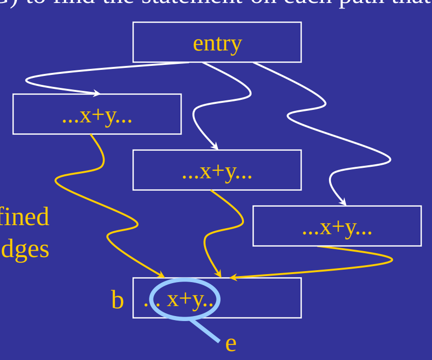
CSE Approach 1

Idea

- If block b uses expression e, and e is available
- Search backward from b (in CFG) to find the statement on each path that most recently generates e
- Insert copy to n after generators
- Replace e with n

Is this a good approach?

x and y not defined along yellow edges



CSE Approach 1 (cont)

Notation

- Avail(b) is the set of expressions available at block b
- Gen(b) is the set of expressions generated and not killed at block b

If we use e and e ∈ Avail(b)

- Allocate a new name n
- Search backward from b (in CFG) to find statement on each path that most recently generates e
- Insert copy to n after generators
- Replace e with n

Problems?

- Backward search for each use is expensive
- Generates unique name for each use
  - |names| ∝ |Uses| > |Avail|
  - Each generator may have many copies

Example

```
    a  := b + c
    t1 := a
    t2 := a
    e  := b + c
b:  f  := b + c
```


CSE Approach 2

Idea

- Reduce number of copies by assigning a unique name to each unique expression

Summary

- ∀e Name[e] = unassigned
- uses: if we use e and e ∈ Avail(b)
  - if Name[e]=unassigned, allocate new name n and Name[e] = n else n = Name[e]
  - Replace e with n
- defs: In a subsequent traversal of block b, if e ∈ Gen(b) and Name[e] ≠ unassigned, then insert a copy to Name[e] after the generator of e

Problem

- Requires two passes over the code
- May still insert unnecessary copies

CSE Approach 3

Idea
- Don’t worry about temporaries
- Create one temporary for each unique expression
- Let subsequent pass eliminate unnecessary temporaries

At an evaluation of e
- Hash e to a name, n, in a table
- Insert an assignment of e to n

At a use of e in b, if e ∈ Avail(b)
- Lookup e’s name in the hash table (call this name n)
- Replace e with n

Problems
- Inserts more copies than approach 2 (but extra copies are dead)
- Still requires two passes (2^{nd} pass is very general)

Comparing the Three Approaches

Approach 1 and Approach 2
- Make decisions about when to insert temporaries
- Approach 1:
  - Insert temporaries as we look for redundant expressions
  - One temporary per use of redundant expression
- Approach 2
  - Use a second pass to insert temporaries

Approach 3
- Don’t worry about temporaries!

Extraneous Copies

Extraneous copies degrade performance

Let other transformations deal with them

- Dead code elimination
- Copy propagation
  - Coalesce assignments to `t1` and `t2` into a single statement

```
t1 := b + c
t2 := t1
```

- Greatly simplifies CSE

Loop Invariant Code Motion

Last Time

- Loop invariant code motion
- Value numbering

Today

- Finish value numbering
- More reuse optimization
  - Common subexpression elimination
  - Partial redundancy elimination

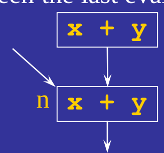
Next Time

- Something special

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



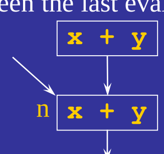
Question

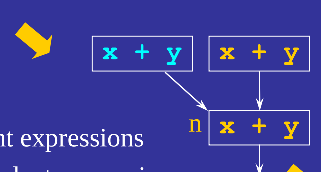
- Can we remove partially redundant code?
- Yes. It's a three step process

Partial Redundancy Elimination (PRE)

Partial Redundancy

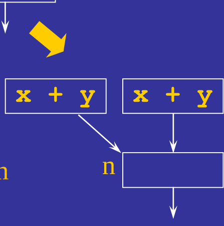
- An expression (*e.g.*, **x+y**) is **partially redundant** at node n if **some** path from the entry node to n evaluates **x+y**, and there are no definitions of **x** or **y** between the last evaluation of **x+y** and n





Three Steps

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



Is this beneficial?

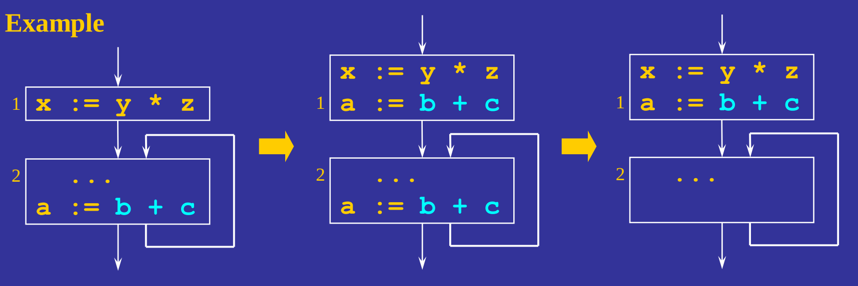
- PRE subsumes CSE and loop invariant code motion

Loop Invariance Example

PRE removes loop-invariant code

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

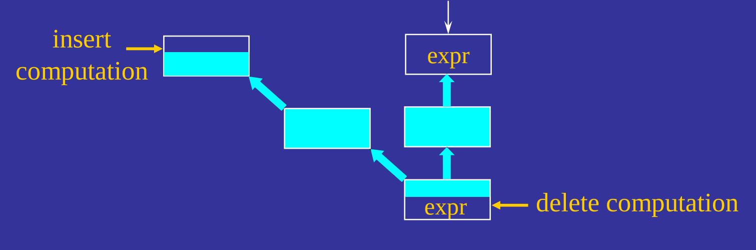
Example



Implementing PRE

Big picture

- Use local properties (**available** and **anticipated**) 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`}

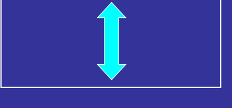
Questions?

Local Properties (cont)

How are these properties useful?

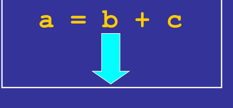
– They tell us where we can introduce redundancy

Transparent



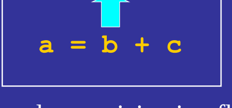
The expression can be redundantly evaluated anywhere in the block

Available



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

Anticipated



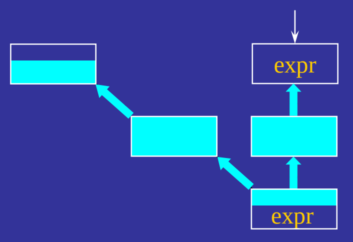
The expression can be redundantly evaluated anywhere before its first evaluation in the block

– In which direction does anticipation flow?

Local Properties (cont)

Example

- For each block in the following figure, what are the local properties with respect to expr?

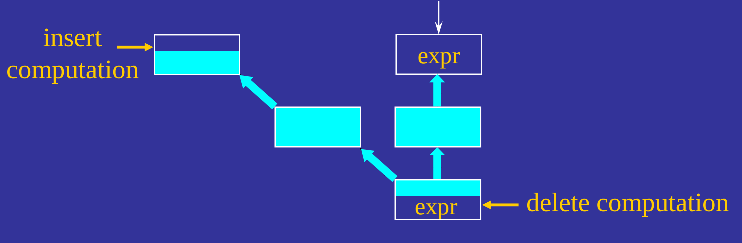


- Where can e be computed redundantly?

Implementing PRE

Big picture

- Use local properties (**available** and **anticipated**) 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



Global Analysis for PRE

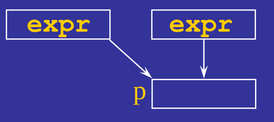
We'll need three global analyses

- Globally available
- Partially available
- Globally anticipated

Globally Available

Intuition

- **Globally available** 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



Data-flow Equations

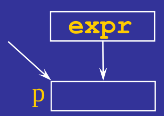
$$\text{available\_in}[n] = \bigcap_{p \in pred[n]} \text{available\_out}[p]$$

$$\text{available\_out}[n] = \boxed{\text{locally\_available}[n]} \cup (\text{available\_in}[n] \cap \boxed{\text{transparent}[n]})$$

(Globally) Partially Available

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



Data-flow Equations?

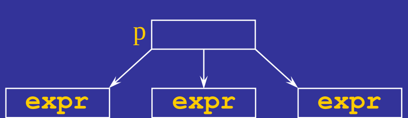
$$\text{partially\_available\_in}[n] = \bigcup_{p \in pred[n]} \text{partially\_available\_out}[p]$$

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

Globally Anticipated

Intuition

- If e is **globally anticipated** at p, then adding an evaluation of e at p will make e redundant along all paths from p, ie, you're expecting e to be computed in the future



Data-flow Equations?

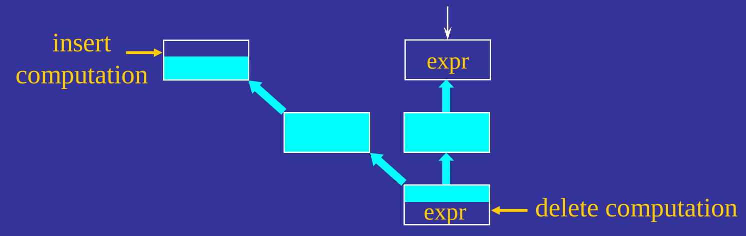
$$\text{anticipated\_out}[n] = \bigcap_{s \in succ[n]} \text{anticipated\_in}[s]$$

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

Implementing PRE

Big picture

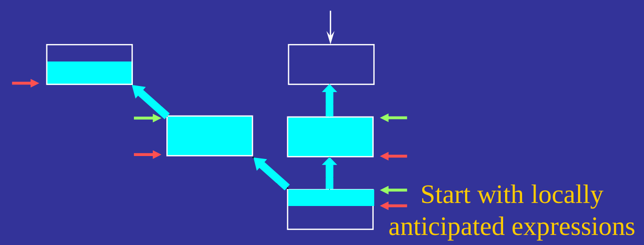
- Use local properties (**available** and **anticipated**) to determine where redundancy can be created within a basic block
- Use global analysis (data-flow analysis) to discover where partial redundancy can be converted to full redundancy
- Insert code and remove redundant expressions



Global Possible Placement

Goal

- Convert partial redundancies to full redundancies How?
- **Possible Placement** is a backwards analysis that identifies locations where such conversions can take place
  - $e \in ppin[n]$ can be placed at entry of n
  - $e \in ppout[n]$ 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

Data-flow Equations

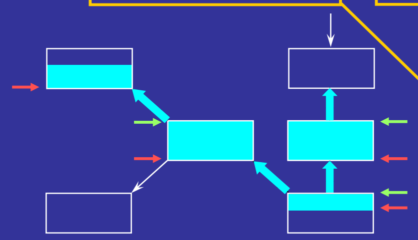
Will turn partial redundancy into full redundancy

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

$$\text{ppin}[n] = \boxed{\text{anticipated\_in}[n]} \cap \boxed{\text{partially\_available\_in}[n]} \cap \boxed{(\text{locally\_anticipated}[n]} \cup \boxed{(\text{ppout}[n] \cap \text{transparent}[n]))}$$

Middle of chain

This block is at the beginning of a chain



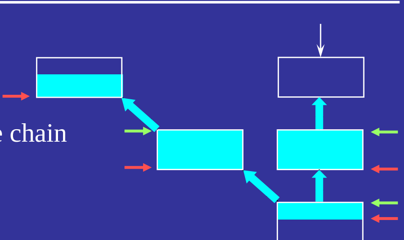
How do we ensure that it is redundant on every edge out of the block?

Updating Blocks

Intuition

– Perform insertion at tops of the chain

– Perform deletion at the bottoms of the chain



Data-flow Equations

– $\text{insert}[n] = \text{ppout}[n] \cap \boxed{(\neg\text{ppin}[n] \cup \neg\,\text{transparent}[n])} \cap \boxed{\neg\text{available\_out}[n]}$

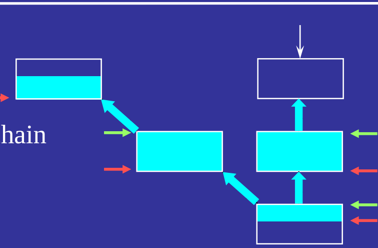
Don’t insert it where it’s fully redundant

– $\text{delete}[n] = \boxed{\text{ppin}[n] \cap \text{locally\_anticipated}[n]}$

Updating Blocks (cont)

Intuition

- Perform insertion at tops of the chain
- Perform deletion at the bottoms of the chain



Functions

- insert[n] = ppout[n]

  ∩ (¬ppin[n] ~~∪ ¬ transparent[n]~~) Can we omit this clause?

  ∩ ¬available_out[n] ¬ ppout[n] ? No

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

February 25, 2015 Reuse Optimization II 31

Exercise

```
B1: a := b + c            B2: b := b + 1

           B3: a := b + c
```

	B1	B2	B3
transparent			
locally_available			
locally_anticipated			
available_in			
available_out			
partially_available_in			
partially_available_out			
anticipated_out			
anticipated_in			
ppout			
ppin			
insert			
delete			

February 25, 2015 Reuse Optimization II 32

Comparing Redundancy Elimination

Value numbering
- Examines values not expressions
- Symbolic
- Knows nothing about algebraic properties (1+x = x+1)

CSE
- Examines expressions

PRE
- Examines expressions
- Subsumes CSE and loop invariant code motion
- Simpler 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

Lecture

– Pointer analysis

Assignment3

– Now available– start early!