Lazy Code Motion

This lecture is primarily based on Konstantinos Sagonas set of slides
(**Advanced Compiler Techniques**, (2AD518)
at Uppsala University, January-February 2004).
Used with kind permission.
(In turn based on Keith Cooper's slides)

Lazy Code Motion

The concept

- Solve data-flow problems that reveal limits of code motion
- Compute INSERT & DELETE sets from solutions
- Linear pass over the code to rewrite it (using INSERT & DELETE)

The history

- Partial redundancy elimination (Morel & Renvoise, CACM, 1979)
- Improvements by Drechsler & Stadel, Joshi & Dhamdhere, Chow, Knoop, Ruthing & Steffen, Dhamdhere, Sorkin, …
- All versions of PRE optimize placement
  - Guarantee that no path is lengthened
- LCM was invented by Knoop et al. in PLDI, 1992
- We will look at a variation by Drechsler & Stadel

SIGPLAN Notices, 28(5), May, 1993

Lazy Code Motion

The intuitions

- Compute *available expressions*
- Compute *anticipable expressions*
- These lead to an earliest placement for each expression
- Push expressions down the CFG until it changes behavior

Assumptions

- Uses a lexical notion of identity (*not value identity*)
- Code is in an Intermediate Representation with unlimited name space
- Consistent, disciplined use of names
  - Identical expressions define the same name
  - No other expression defines that name

} **Avoids copies**
Result serves as proxy

Lazy Code Motion

The Name Space

- $r_i + r_j \rightarrow r_k$, always (*hash to find k*)
- We can refer to $r_i + r_j$ by r_k (*bit-vector sets*)
- Variables must be set by copies
  - No consistent definition for a variable
  - Break the rule for this case, but require $r_{source} < r_{destination}$
  - To achieve this, assign register names to variables first

Without this name space

- LCM must insert copies to preserve redundant values
- LCM must compute its own map of expressions to unique ids

Lazy Code Motion: Running Example

$\mathcal{B}_1$:

$r_1 \leftarrow 1$

$r_2 \leftarrow r_1$

$r_3 \leftarrow r_0 + @m$

$r_4 \leftarrow r_3$

$r_5 \leftarrow (r_1 < r_2)$

if r_5 then $\mathcal{B}_2$ else $\mathcal{B}_3$

$\mathcal{B}_2$:

$r_{20} \leftarrow r_{17} * r_{18}$

$r_{21} \leftarrow r_{19} + r_{20}$

$r_8 \leftarrow r_{21}$

$r_6 \leftarrow r_2 + 1$

$r_2 \leftarrow r_6$

$r_7 \leftarrow (r_2 > r_4)$

if r_7 then $\mathcal{B}_3$ else $\mathcal{B}_2$

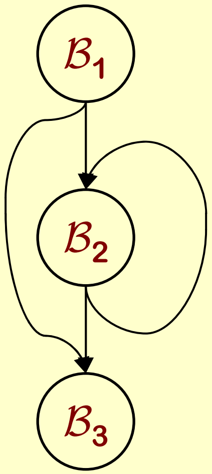
$\mathcal{B}_3$: ...

Variables:

r_2, r_4, r_8

Expressions:

$r_1, r_3, r_5, r_6, r_7, r_{20}, r_{21}$



Lazy Code Motion

Predicates (computed by Local Analysis)

- **DEEXPR**(b) contains expressions defined in b that survive to the end of b.

  e ∈ **DEEXPR**(b) ⇒ evaluating e at the end of b produces the same value for e as evaluating it in its original position.

- **UEEXPR**(b) contains expressions defined in b that have upward exposed arguments (both args).

  e ∈ **UEEXPR**(b) ⇒ evaluating e at the start of b produces the same value for e as evaluating it in its original position.

- **KILLEDEXPR**(b) contains those expressions whose arguments are (re)defined in b.

  e ∈ **KILLEDEXPR**(b) ⇒ evaluating e at the start of b does not produce the same result as evaluating it at its end.

Lazy Code Motion: Running Example

```
B1:
   r1←1
   r2←r1
   r3←r0+@m
   r4←r3
   r5←(r1< r2)
   if r5 then B2 else B3
B2:
   r20←r17*r18
   r21←r19+r20
   r8←r21
   r6←r2+1
   r2←r6
   r7←(r2>r4)
   if r7 then B3 else B2
B3:  ...
```

Variables:

r_2, r_4, r_8

Expressions:

$r_1, r_3, r_5, r_6, r_7, r_{20}, r_{21}$

	$\mathcal{B}1$	$\mathcal{B}2$	$\mathcal{B}3$
DEEXPR	r1, r3, r5	r7, r20, r21	
UEEXPR	r1, r3	r6, r20	
KILLEDEXPR	r5, r6, r7	r5, r6, r7, r21	

Lazy Code Motion

Availability

$$\text{AVAILIN}(n) = \cap_{m \in preds(n)} \text{AVAILOUT}(m), \qquad n \neq n_0$$

$$\text{AVAILOUT}(m) = \text{DEEXPR}(m) \cup (\text{AVAILIN}(m) \cap \overline{\text{KILLEDEXPR}(m)})$$

Initialize AVAILIN(n) to the set of all names, except at n_0

Set AVAILIN(n_0) to $\varnothing$

Interpreting AVAIL

- e $\in$ AVAILOUT(b) $\Leftrightarrow$ evaluating e at end of b produces the same value for e. AVAILOUT tells the compiler how far forward e can move the evaluation of e, ignoring any uses of e.
- This differs from the way we talk about AVAIL in global redundancy elimination.

Lazy Code Motion

Anticipability

$$\text{ANTOUT}(n) = \bigcap_{m \in succs(n)} \text{ANTIN}(m), \quad n \text{ not an exit block}$$

$$\text{ANTIN}(m) = \text{UEEXPR}(m) \cup (\text{ANTOUT}(m) \cap \overline{\text{KILLEDEXPR}(m)})$$

Initialize ANTOUT(n) to the set of all names, except at exit blocks

Set ANTOUT(n) to ∅, for each exit block n

Interpreting ANTOUT

- e ∈ ANTIN(b) ⇔ evaluating e at start of b produces the same value for e. ANTIN tells the compiler how far backward e can move
- This view shows that anticipability is, in some sense, the inverse of availability (& explains the new interpretation of AVAIL).

Lazy Code Motion

Earliest placement

$$\text{EARLIEST}(i,j) = \text{ANTIN}(j) \cap \overline{\text{AVAILOUT}(i)} \cap (\text{KILLEDEXPR}(i) \cup \overline{\text{ANTOUT}(i)})$$

$$\text{EARLIEST}(n_0,j) = \text{ANTIN}(j) \cap \overline{\text{AVAILOUT}(n_0)}$$

EARLIEST is a predicate

- Computed for edges rather than nodes *(placement)*
- e ∈ EARLIEST(i,j) if
  - It can move to head of j,
  - It is not available at the end of i, and
    - either it cannot move to the head of i (KILLEDEXPR(i))
    - or another edge leaving i prevents its placement in i ($\overline{\text{ANTOUT}(i)}$)

Lazy Code Motion

Later (than earliest) placement

$$\text{LATERIN}(j) = \cap_{i \in preds(j)} \text{LATER}(i,j), \quad j \neq n_0$$

$$\text{LATER}(i,j) = \text{EARLIEST}(i,j) \cup (\text{LATERIN}(i) \cap \overline{\text{UEEXPR}(i)})$$

Initialize **LATERIN**(n_0) to ∅

x ∈ **LATERIN**(k) ⇔ every path that reaches k has x ∈ **EARLIEST**(m) for some block m, and the path from m to k is x-clear & does not evaluate x.

⇒ the compiler can move x through k without losing any benefit.

x ∈ **LATER**(i,j) ⇔ <i,j> is its earliest placement, or it can be moved forward from i (**LATER**(i)) and placement at entry to i does not anticipate a use in i (*moving it across the edge exposes that use*).

Lazy Code Motion

Rewriting the code

$$\text{INSERT}(i,j) = \text{LATER}(i,j) \cap \overline{\text{LATERIN}(j)}$$

$$\text{DELETE}(k) = \text{UEEXPR}(k) \cap \overline{\text{LATERIN}(k)},\ k \neq n_0$$

INSERT & **DELETE** are predicates

Compiler uses them to guide the rewrite step

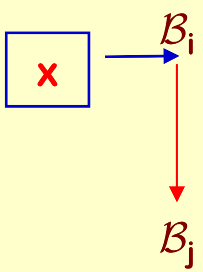
- $x \in$ **INSERT**$(i,j) \Rightarrow$ insert x at start of i, end of j, or new block
- $x \in$ **DELETE**$(k) \Rightarrow$ delete first evaluation of x in k

If local redundancy elimination has already been performed, only one copy of **x** exists. Otherwise, remove all upward exposed copies of **x**.

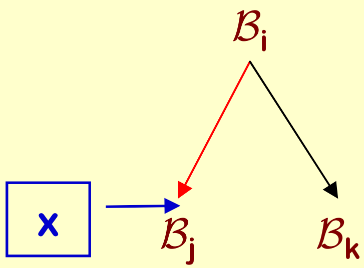
Lazy Code Motion

Edge placement

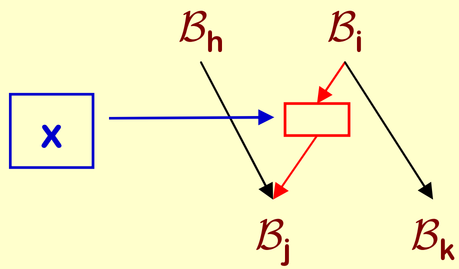
- x ∈ INSERT(i,j)



|*succs*(i)| = 1



|*preds*(j)| = 1



|*succs*(i) > 1

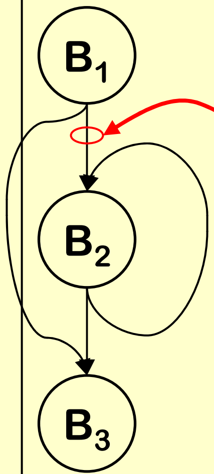
|*preds*(j)| > 1

Three cases

- |*succs*(i)| = 1 ⇒ insert x at end of i.
- |*succs*(i)| > 1 but |*preds*(j)| = 1 ⇒ insert x at start of j.
- |*succs*(i)| > 1 and |*preds*(j)| > 1 ⇒ create new block in <i,j> for x.

Lazy Code Motion Example

B_1: $r_1 \leftarrow 1$
  $r_2 \leftarrow r_1$
  $r_3 \leftarrow r_0 + @m$
  $r_4 \leftarrow r_3$
  $r_5 \leftarrow (r_1 < r_2)$
  if r_5 then B_2 else B_3
B_2: $r_{20} \leftarrow r_{17} * r_{18}$
  $r_{21} \leftarrow r_{19} + r_{20}$
  $r_8 \leftarrow r_{21}$
  $r_6 \leftarrow r_2 + 1$
  $r_2 \leftarrow r_6$
  $r_7 \leftarrow (r_2 > r_4)$
  if r_7 then B_3 else B_2
B_3: ...



	B1	B2	B3
DEEXPR	r1, r3, r5	r7, r20, r21	
UEEXPR	r1, r3	r6, r20	
KILLEDEXPR	r5, r6, r7	r5, r6, r7,r21	

	B1	B2	B3
AVAILIN	{}	r1, r3	r1, r3
AVAILOUT	r1, r3, r5	r1, r3, r7, r20, r21	…
ANTIN	r1, r3	r6, r20	{}
ANTOUT	{}	{}	{}

	1,2	1,3	2,2	2,3
EARLIEST	r20, r21	{}	{}	{}

Example is too small to show off LATER

INSERT(1,2) = { r_{20}, r_{21} }

DELETE(2) = { r_{20} , r_{21} }