COMP 512
Rice University
Spring 2015

# *Lazy Code Motion*

## *— The Data-Flow Approach to Code Motion —*

J. Knoop, O. Ruthing, & B. Steffen, "Lazy Code Motion", in Proceedings of the ACM SIGPLAN 92 Conference on Programming Language Design and Implementation, June 1992. [225]

K. Drechsler & M. Stadel, "A Variation of Knoop, Ruthing, and Steffen's Lazy Code Motion," SIGPLAN Notices, 28(5), May 1993. [134]

§ 10.3.1 of EaC2e



Citation numbers refer to entries in the EaC2e bibliography.

## Announcements

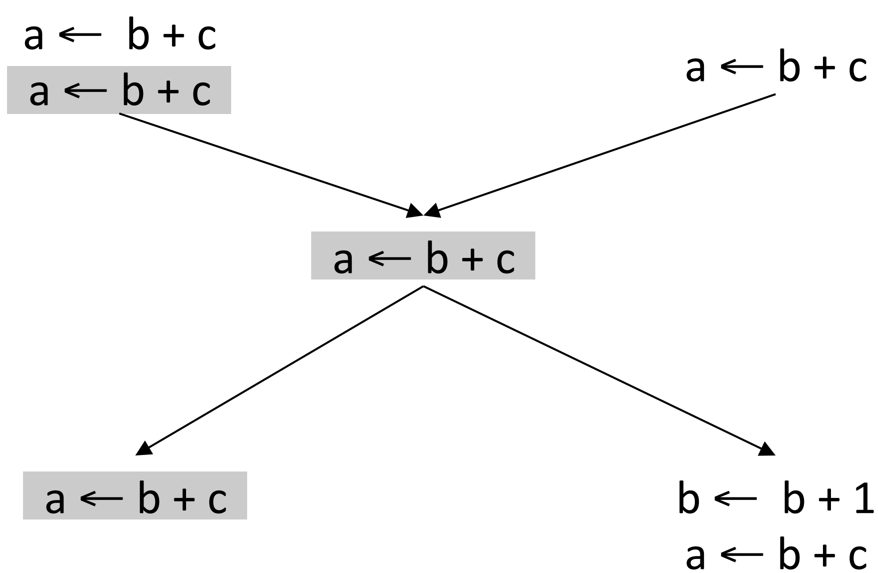
- Next week is break
- Midterm exam
  - ♦ Available today, in class
  - ♦ Due back on Tuesday 3/10/2015 at 5 **PM**
  - ♦ Three questions:
    - → Matching question through today's lecture
    - → Question on data-flow analysis
    - → Question on the construction of **SSA** form
  - ♦ Two-hours, closed-notes, closed-literature, take-home exam
- You should be working on your labs
  - ♦ Three benchmarks available on **CLEAR**
  - ♦ More to come

# Redundant Expression

An expression is redundant at point *p* if, on every path to *p*

1. It is evaluated before reaching *p*, and
2. Non of its constituent values is redefined before *p*

## Example

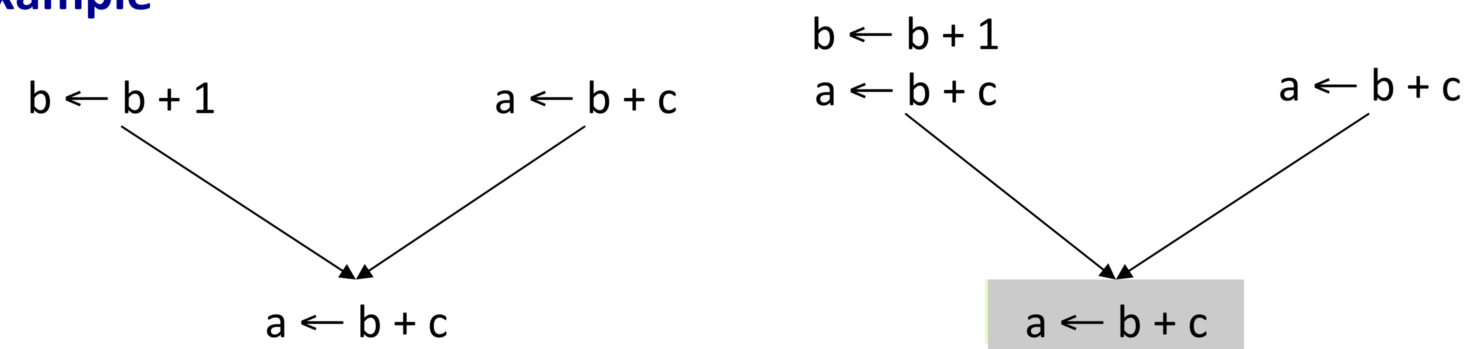



Some occurrences of b+c are redundant

# Partially Redundant Expression

An expression is partially redundant at *p* if it is redundant along some, but not all, paths reaching *p*

**Example**




Inserting a copy of “a ← b + c” after the definition of b can make it redundant

fully redundant?

## Loop Invariant Expression

### Another Example


### Loop invariant expressions are partially redundant

- Partial redundancy elimination performs code motion
- Major part of the work is figuring out where to insert operations

# Lazy Code Motion

## The Concept

- Solve data-flow problems that show opportunities & limits
  - ♦ Availability & anticipability, then placement
- Compute **INSERT** & **DELETE** sets from solutions
- Linear pass over the code to rewrite it (using **INSERT** & **DELETE**)

## The History

- Partial redundancy elimination (**PRE**) [267] (*Morel & Renvoise, CACM, 1979*)
- Improvements by Drechsler & Stadel [133], Joshi & Dhamdhere [209], Chow [81], Knoop, Ruthing & Steffen [225], Dhamdhere [130], Sorkin [321], Hailperin [178], Kennedy, Lo, et al. [220] …
- All versions of **PRE** optimize placement
  - ♦ Guarantee that no path is lengthened
- **LCM** was published by Knoop et al. in **PLDI 92**
- Drechsler & Stadel simplified the equations

**PRE** and its descendants are conservative

Dhamdhere applied these same ideas to strength reduction [127, 131] and hoisting [129]. Others have followed this path, as well [209, 220, 226].

# Lazy Code Motion

**LCM** operates on expressions

It moves expression evaluations, not assignments

## The Intuitions

- Compute *available expressions*
- Compute *anticipable expressions*
- From **AVAIL** & **ANT**, we can compute 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*)
- ILOC-style code with unlimited name space
- Consistent, disciplined use of names
  - ♦ Identical expressions define the same name
  - ♦ No other expression defines that name

} Avoids copies; Result name serves as proxy

**LCM** operates on code that is *not* in **SSA** form.
Lexical identity conflicts with **SSA**'s notion of unique names.

## Lazy Code Motion

Digression in Chapter 5 of EAC2e:
"The impact of naming"

### The Name Space

- $r_i + r_j \rightarrow r_k$, always, with both i < k and j < k (*hash to find k*)
  - ♦ $r_k$ is always set by $r_i + r_j$ or $r_j + r_i$, and by no other expression
- 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

The restrictions on the name space in **LCM** goes all the way back to Morel & Renvoise [267]. It is mentioned as an assumption in the original paper.

# Lazy Code Motion

## Local Information

*(Computed for each block)*

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

  *(downward exposed expressions)*

  e ∈ **DEEXPR**(b) ⇒ evaluating e at the end of b produces the same value for e

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

  e ∈ **UEEXPR**(b) ⇒ evaluating e at the start of b produces the same value for e

- **EXPRKILL**(b) contains those expressions that have one or more arguments defined (*killed*) in b *(killed expressions)*

  e ∉ **EXPRKILL**(b) ⇒ evaluating e produces the same result at the start and end of b

# Lazy Code Motion

## Availability

$$\textbf{AVAILIN}(n) = \bigcap_{m \in preds(n)} \textbf{AVAILOUT}(m), \quad n \neq n_0$$

$$\textbf{AVAILOUT}(m) = \textbf{DEEXPR}(m) \cup (\textbf{AVAILIN}(m) \cap \overline{\textbf{EXPRKILL}(m)})$$

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

Set **AVAILIN**($n_0$) to Ø

## Interpreting AVAILOUT

- e ∈ **AVAILOUT**(b) ⇔ evaluating e at end of b produces the same value for e. **AVAILOUT** tells the compiler how far forward e can move
- This interpretation differs from the way we *talk* about **AVAILOUT** in global redundancy elimination; the equations, however, are unchanged.

## Lazy Code Motion

Anticipability is identical to VeryBusy expressions

### Anticipability

$$\textbf{ANTOUT}(n) = \cap_{m \in succs(n)} \textbf{ANTIN}(m), \qquad n \text{ not an exit block}$$

$$\textbf{ANTIN}(m) = \textbf{UEEXPR}(m) \cup (\textbf{ANTOUT}(m) \cap \overline{\textbf{EXPRKILL}(m)})$$

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

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

### Interpreting ANTOUT

- $e \in$ **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 availablilty (& explains the new interpretation of **AVAIL**)

# Lazy Code Motion

## The Intuitions

*Available expressions*

- $e \in$ **AVAILOUT**($b$) ⇒ evaluating *e* at exit of b gives same result
- $e \in$ **AVAILIn**($b$) ⇒ *e* is available from every predecessor of *b*

  ⇒ an evaluation at entry of *b* is redundant

*Anticipable expressions*

- $e \in$ **ANTIN**($b$) ⇒ evaluating *e* at entry of b gives same result
- $e \in$ **ANTOUT**($b$) ⇒ *e* is anticipable from every successor of *b*

  ⇒ evaluation at exit of *b* would a later evaluation redundant, on every path, so exit of *b* is a profitable place to insert *e*

# Lazy Code Motion

## Earliest Placement On An Edge

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

Can move *e* to head of *j* & it is not redundant from *i*
*and*
Either killed in *i* or would not be busy at exit of *i*

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

⇒ insert *e* on the edge

**EARLIEST** is a predicate

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

# Lazy Code Motion

## Later (than earliest) Placement

$$\textbf{LATERIN}(j) = \bigcap_{i \in pred(j)} \textbf{LATER}(i,j), \qquad j \neq n_0$$

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

Initialize **LATERIN**($n_0$) to Ø

x ∈ **LATERIN**(k) ⇔ every path that reaches k has x ∈ **EARLIEST**(i,j) for some edge (i,j) leading to x, and the path from the entry of j 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*)

Propagate forward until a block kills it ($\overline{\textbf{UEEXPR}}$)

# Lazy Code Motion

## Rewriting the code

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

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

Can go on the edge but not in j ⇒ no later placement

Upward exposed (so we will cover it) & not an evaluation that might be used later

**INSERT** & **DELETE** are predicates

Compiler uses them to guide the rewrite step

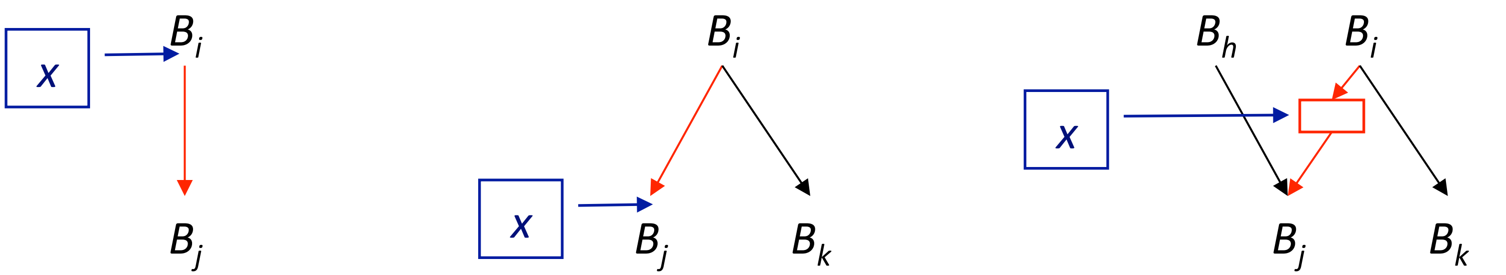
- *x* ∈ **INSERT**(i,j) ⇒ insert *x* at start of j, end of i, or new block
- *x* ∈ **DELETE**(k) ⇒ delete first evaluation of *x* in k [1]

[1] 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

- $x \in$ **INSERT**(i,j)




|*succs*(i)| = 1

|*preds*(j)| = 1

|*succs*(i) > 1 & |*preds*(j)| > 1

**A "critical" edge**

## 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, & |*preds*(j)| > 1 ⇒ create new block in <i,j> for *x*

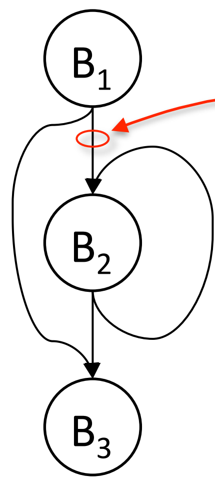
# Lazy Code Motion

## Example

$B_1$: $r_1 \leftarrow 1$
$r_2 \leftarrow r_0 + @m$
if $r_1 < r_2 \rightarrow B_2, B_3$

$B_2$: ...
$r_{20} \leftarrow r_{17} * r_{18}$
...
$r_4 \leftarrow r_1 + 1$
$r_1 \leftarrow r_4$
if $r_1 < r_2 \rightarrow B_2, B_3$

$B_3$: ...




| | B1 | B2 |
|---|---|---|
| **DEEXPR** | r1,r2 | r1,r4,r20 |
| **UEEXPR** | r1,r2 | r4,r20 |
| **NotKilled** | r17,r18,r20 | r2,r17,r18,r20 |

| | B1 | B2 |
|---|---|---|
| **AVAILIN** | r17,r18 | r1,r2,r17,r18 |
| **AVAILOUT** | r1,r2,r17,r18 | r1,r2,r4,r17,r18,r20 |
| **ANTIN** | { } | r20 |
| **ANTOUT** | { } | { } |

| | 1,2 | 1,3 | 2,2 | 2,3 |
|---|---|---|---|---|
| **EARLIEST** | r20 | { } | { } | { } |

Critical edge rule will create landing pad when needed, as on edge $(B_1, B_2)$

Example is too small to show off **LATER**

**INSERT**(1,2) = { $r_{20}$ }

**DELETE**(2) = { $r_{20}$ }

See the papers for more detailed examples.

# Lazy Code Motion

## Improving the Results

Simpson attacked the problem of **LCM**'s reliance on lexical identity

- Performed global value numbering, then rewrote the name space to encode value identity into lexical identity
- In essence, his technique joined the code placement aspects of **LCM** with the value-based equivalence detection of global value numbering

Briggs rearranged expressions to expose more lexical identities

- Used algebraic reassociation to rewrite expressions into a canonical form
  - ♦ Associativity & commutativity, + distribution in some limited forms
- Preconditioning the code with reassociation exposed more opportunities