Comp 512
Spring 2011

Lazy Code Motion

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

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

§ 10.3.1 of EaC2e

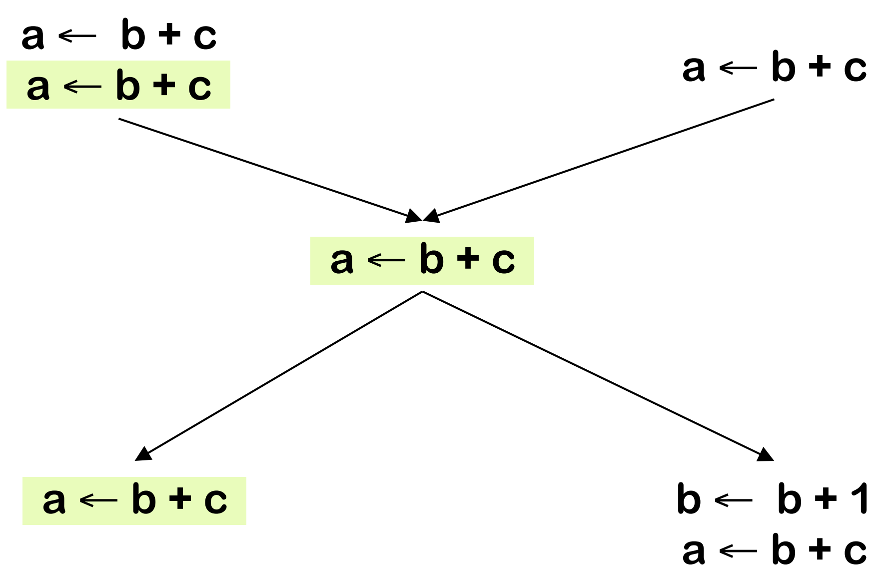
Drechsler & Stadel give a much more complete and satisfying example.

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 constitutent values is redefined before *p*

Example

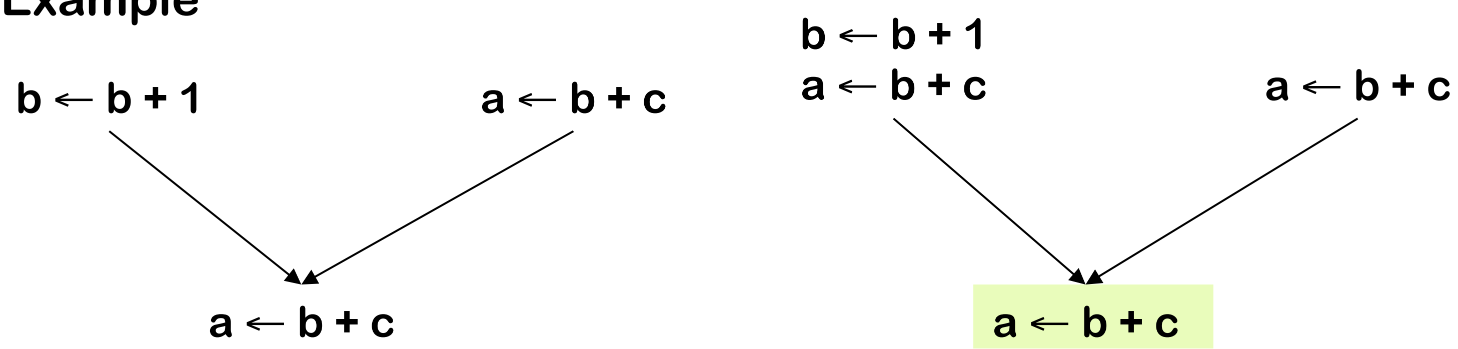


Some occurrences of b+c are redundant

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
- 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

PRE and its descendants are conservative

- LCM was invented by Knoop et al. in PLDI, 1992
- Drechsler & Stadel simplified the equations

Citations for the critical papers are on the title page.

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 serves as proxy

Digression in Chapter 5 of EAC: “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*)
- 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

LCM operates on expressions

It moves expression evaluations, not assignments

Lazy Code Motion

Local Informtion (*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

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

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

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

Set AVAILIN(n_0) to Ø

Interpreting AVAIL

- **$e \in$ AVAILOUT(b) $\Leftrightarrow$ evaluating e at end of b produces the same value for e. AVAILOUT tells the compiler that an evaluation at the end of the block is covered by the evaluation earlier in the block. It also shows that evaluation of e can move to the end of the block.**
- **This interpretation differs from the way we *talk* about AVAIL in global redundancy elimination; the equations, however, are unchanged.**

Lazy Code Motion

Anticipability is identical to VeryBusy expressions

Anticipability

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

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

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 ∈ ANTIN(b) ⇔ evaluating e at start of b produces the same value for e. ANTIN tells the compiler how far backward e can move. If e is also in AVAILIN(b), the evaluation in the block is redundant.
- 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* ∈ AVAILOUT(*b*) ⇒ evaluating *e* at exit of *b* gives same result
  ⇒ *e* could move to exit of *b*
- *e* ∈ AVAILIN(*b*) ⇒ *e* is available from every predecessor of *b*
  ⇒ an evaluation of *e* at entry of *b* is redundant

Anticipable expressions

- *e* ∈ ANTIN(*b*) ⇒ evaluating *e* at entry of b gives same result
- ⇒ *e* could move to entry of *b*
- *e* ∈ ANTOUT(*b*) ⇒ *e* is used on every path leaving *b*
  ⇒ evaluations in *b*'s successors could move to the end of *b*

Lazy Code Motion

Earliest placement on an edge

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

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*

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

⇒ 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 (EXPRKILL(i))
  - either it cannot move to the head of i or another edge leaving i prevents its placement in i ($\overline{\text{ANTOUT}(i)}$)

Lazy Code Motion

Later (than earliest) placement

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

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

Initialize LATERIN(n_0) to 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 ($\overline{\text{UEEXPR}}$)

Lazy Code Motion

Rewriting the code

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

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

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

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

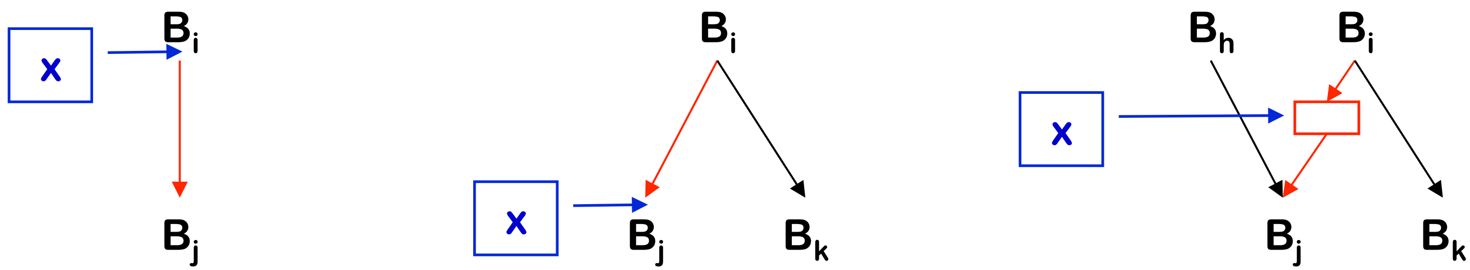
- $x \in$ INSERT(i,j) ⇒ insert x at start of j, end of i, or new block
- $x \in$ DELETE(k) ⇒ delete first evaluation of x in k

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

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



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

A “critical” edge

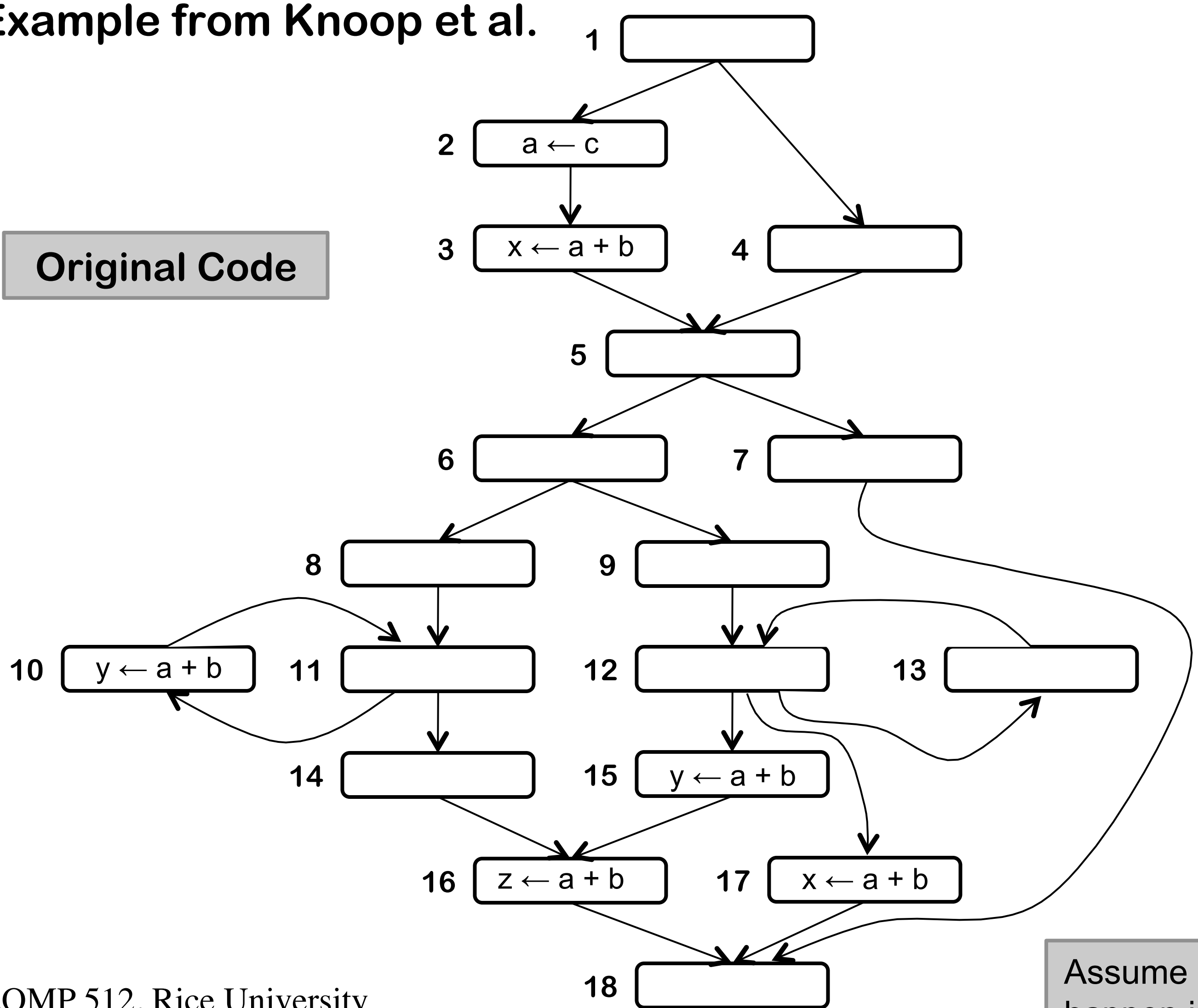
Three cases

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

Lazy Code Motion

Example from Knoop et al.

Original Code

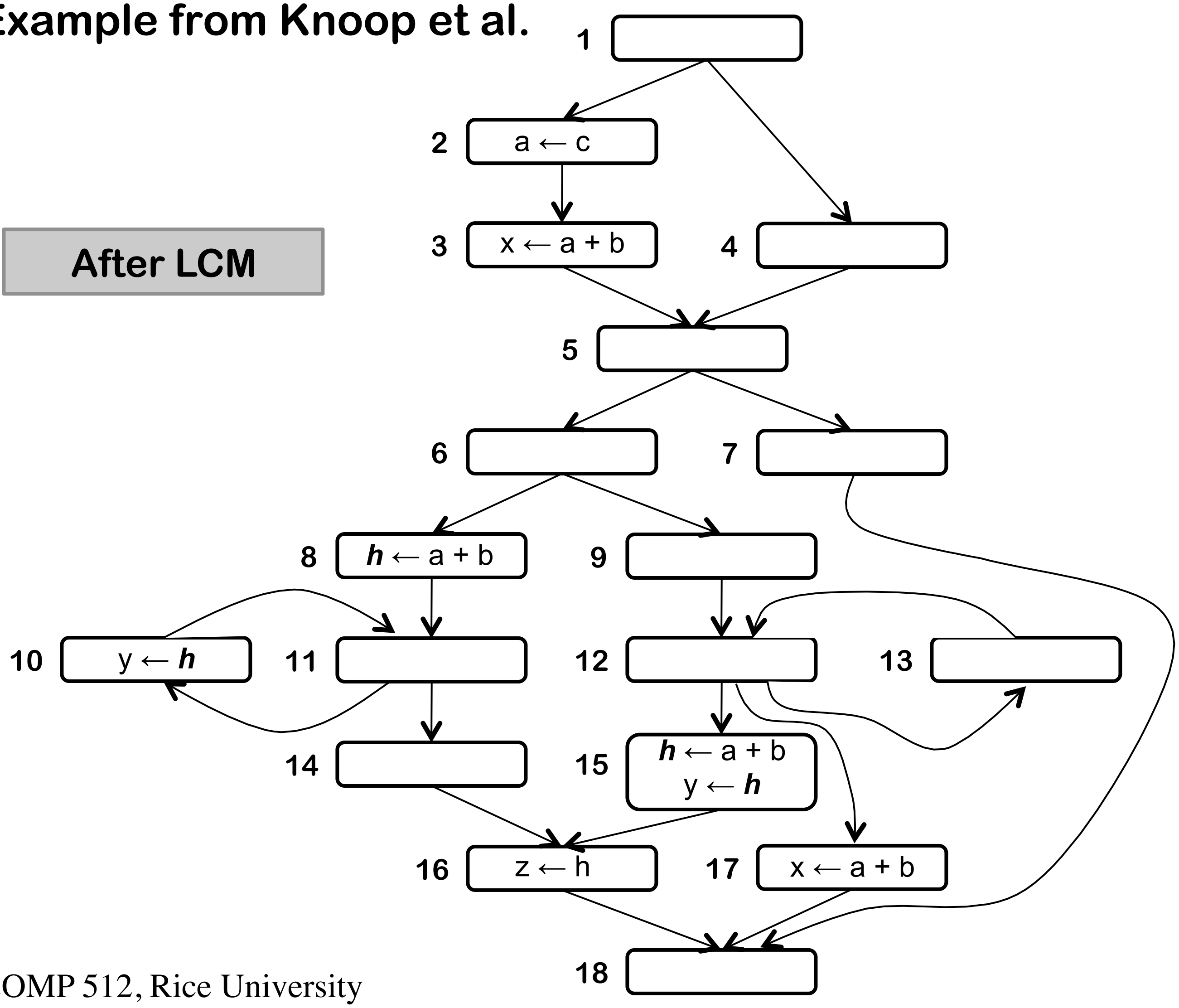


Assume that bad things can happen in an empty box

Lazy Code Motion

Example from Knoop et al.

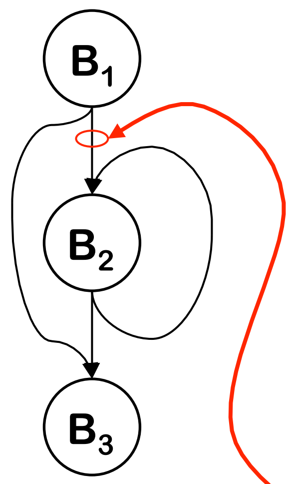
After LCM



Lazy Code Motion

Example

B_1: $r_1 \leftarrow 1$
$\quad r_2 \leftarrow r_0 + @m$
$\quad$ if $r_1 < r_2 \rightarrow B_2, B_3$
B_2: …
$\quad r_{20} \leftarrow r_{17} * r_{18}$
$\quad$ …
$\quad r_4 \leftarrow r_1 + 1$
$\quad r_1 \leftarrow r_4$
$\quad$ 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
$\quad$ Insert(1,2) = { r_{20} }
$\quad$ Delete(2) = { r_{20} }