

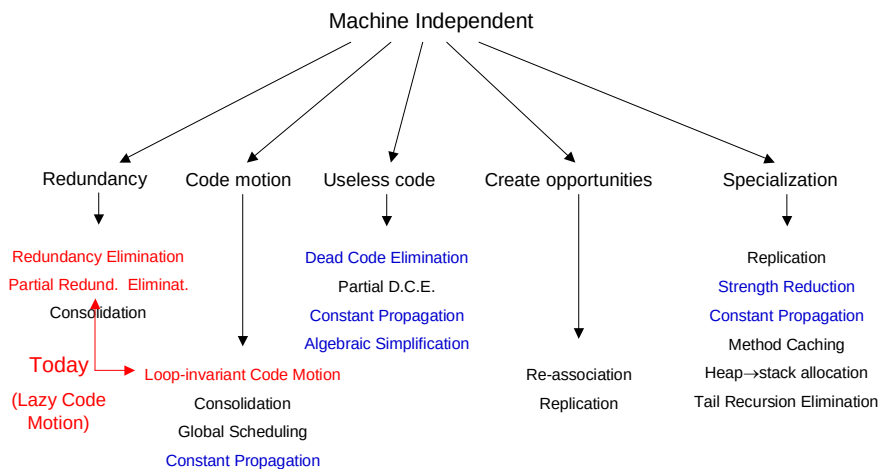
Partial Redundancy Elimination & Lazy Code Motion

The "**Lazy Code Motion**" part of this lecture draws on two papers:

1. "Lazy Code Motion", by J. Knoop, O. Rüthing, and B. Steffen, in PLDI'92;
2. "A Variation of Knoop, Rüthing, and Steffen's Lazy Code Motion", by K. Drechsler and M. Stadel, SIGPLAN Notices, 28(5), May 1993;

and is primarily based on Keith Cooper's set of slides (COMP 512 class at Rice University, Fall 2002). Used with kind permission.

Taxonomy of Global Compiler Optimizations



KT2, 2004

* 2

Common-Subexpression Elimination

An occurrence of an expression in a program is a common subexpression if there is another occurrence of the expression whose evaluation always precedes this one in execution order and if the operands of the expression remain unchanged between the two evaluations.

Local Common Subexpression Elimination (CSE) keeps track of the set of *available expressions* within a basic block and replaces instances of them by references to new temporaries that keep their value.

```
...  
a = (x+y)+z;  
b = a-1;  
c = x+y;  
...
```

Before CSE

```
...  
t = x+y;  
a = t+z;  
b = a-1;  
c = t;  
...
```

After CSE

KT2, 2004

3

Available Expressions

- An expression $x+y$ is available at a program point p if
 - > every path from the initial node to p evaluates $x+y$ before reaching p ,
 - > and there are no assignments to x or y after the evaluation but before p .
- Available Expression information can be used to do global (across basic blocks) CSE.
- If an expression is available at the point of its use, there is no need to re-evaluate it.

KT2, 2004

4

Computing Available Expressions

- Represent sets of expressions using bit vectors
- Each expression corresponds to a bit
- Run dataflow algorithm similar to reaching definitions
- Notice that:
 - > A definition reaches a basic block if it comes from ANY predecessor in CFG.
 - > An expression is available at a basic block only if it is available from ALL block's predecessors in the CFG.

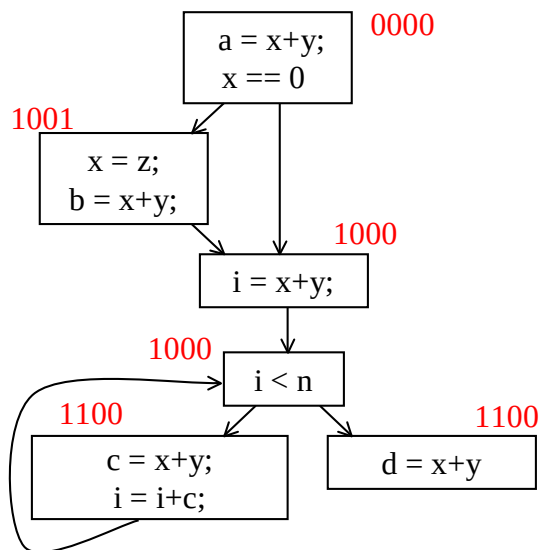
KT2, 2004

5

Global CSE Example

Expressions

- 1: $x+y$
- 2: $i < n$
- 3: $i+c$
- 4: $x==0$



KT2, 2004

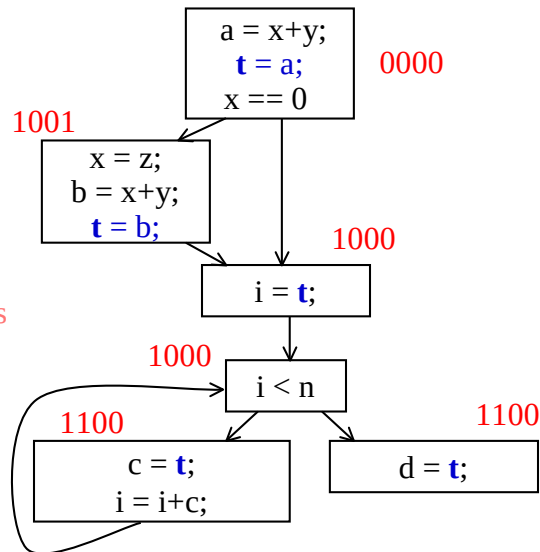
6

Global CSE Transform

Expressions

- 1: $x+y$
- 2: $i < n$
- 3: $i+c$
- 4: $x==0$

must use same temp
for CSE in all blocks



KT2, 2004

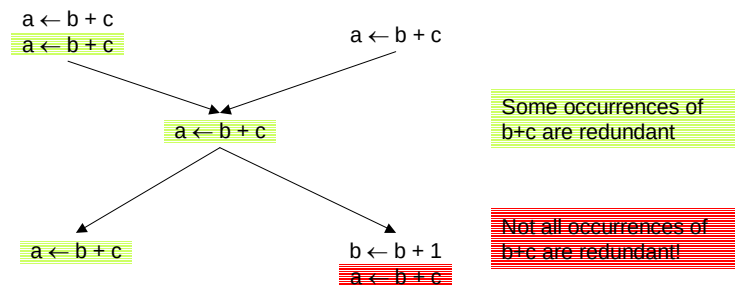
7

Redundant Expressions

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

1. It is evaluated before reaching p , and
2. None of its constituent values is redefined before p

Example



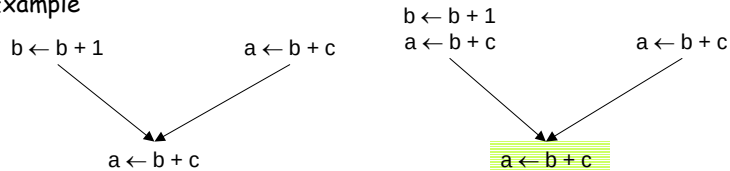
KT2, 2004

* 8

Partially Redundant Expressions

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

Example



Inserting a copy of " $a \leftarrow b + c$ " after the definition of b can make it redundant

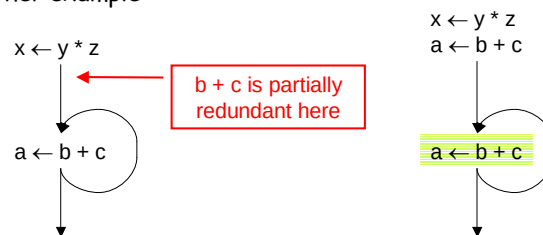
fully redundant?

KT2, 2004

* 9

Loop Invariant Expressions

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

KT2, 2004

10

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

KT2, 2004

11

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

KT2, 2004

12

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

KT2, 2004

13

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: r₂, r₄, r₈
Expressions: r₁, r₃, r₅, r₆, r₇, r₂₀, r₂₁

KT2, 2004

14

Lazy Code Motion

Predicates (computed by Local Analysis)

- $DEEXPR(b)$ contains expressions defined in b that survive to the end of b .
 $e \in DEEXPR(b) \Rightarrow$ 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).
 $e \in UEEXPR(b) \Rightarrow$ 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 .
 $e \in KILLEDEXPR(b) \Rightarrow$ evaluating e at the start of b does not produce the same result as evaluating it at its end.

KT2, 2004

15

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: r₂, r₄, r₈
 Expressions: r₁, r₃, r₅, r₆, r₇, r₂₀, r₂₁

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

KT2, 2004

16

Lazy Code Motion

Availability

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

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

Initialize $\text{AVAILIN}(n)$ to the set of all names, except at n_0

Set $\text{AVAILIN}(n_0)$ to $\emptyset$

Interpreting AVAIL

- $e \in \text{AVAILOUT}(b) \Leftrightarrow$ 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

KT2, 2004

17

Lazy Code Motion

Anticipability

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

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

Initialize $\text{ANTOUT}(n)$ to the set of all names, except at exit blocks

Set $\text{ANTOUT}(n)$ to $\emptyset$, for each exit block n

Interpreting ANTOUT

- $e \in \text{ANTIN}(b) \Leftrightarrow$ 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).

KT2, 2004

18

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*)
- $e \in \text{EARLIEST}(i,j)$ 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 ($\text{KILLEDEXPR}(i)$)
 - or another edge leaving i prevents its placement in i ($\overline{\text{ANTOUT}(i)}$)

KT2, 2004

19

Lazy Code Motion

Later (than earliest) placement

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

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

Initialize $\text{LATERIN}(n_0)$ to $\emptyset$

- $x \in \text{LATERIN}(k) \Leftrightarrow$ every path that reaches k has $x \in \text{EARLIEST}(m)$ for some block m , and the path from m to k is x -clear & does not evaluate x
 - $\Rightarrow$ the compiler can move x through k without losing any benefit
- $x \in \text{LATER}(i,j) \Leftrightarrow \langle i,j \rangle$ is its earliest placement, or it can be moved forward from i ($\text{LATER}(i)$) and placement at entry to i does not anticipate a use in i (*moving it across the edge exposes that use*)

KT2, 2004

20

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

- $x \in \text{INSERT}(i,j) \Rightarrow$ insert x at start of i , end of j , or new block
- $x \in \text{DELETE}(k) \Rightarrow$ 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 .

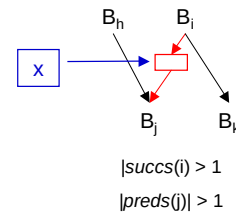
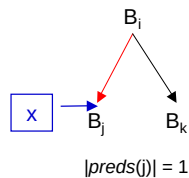
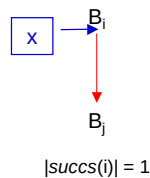
KT2, 2004

*21

Lazy Code Motion

Edge placement

- $x \in \text{INSERT}(i,j)$



Three cases

- $|succs(i)| = 1 \Rightarrow$ insert at end of i
- $|succs(i)| > 1$ but $|preds(j)| = 1 \Rightarrow$ insert at start of j
- $|succs(i)| > 1$ and $|preds(j)| > 1 \Rightarrow$ create new block in $\langle i,j \rangle$ for x

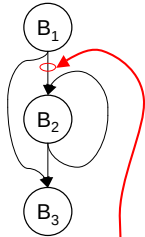
KT2, 2004

22

Lazy Code Motion 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: ...
    
```



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

	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) = { r₂₀, r₂₁ }

DELETE(2) = { r₂₀, r₂₁ }