Partial Redundancy Elimination, Loop Optimization & Lazy Code Motion

Advanced Compiler Techniques
2005
Erik Stenman
Virtutech

Common-Subexpression Elimination
(Repetition)

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;
…
```

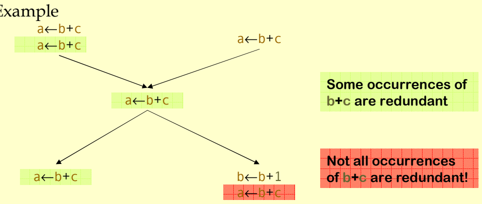
After CSE

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*

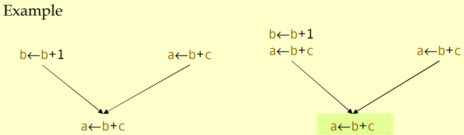
Example



Partially Redundant Expressions

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

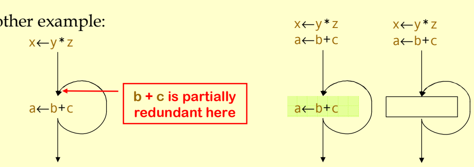
Example



Inserting a copy of "a←b+c" after the definition of b can make it redundant.

Loop Invariant Expressions

Another example:



Loop invariant expressions are partially redundant.

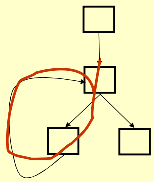
- Partial redundancy elimination performs code motion.
- Major part of the work is figuring out where to insert operations.

Loop Optimizations

- Loop Optimization is important because most of the execution time occurs in loops.
- First, we will identify loops.
- We will study three optimizations:
  - Loop-invariant code motion.
  - Strength reduction.
  - Induction variable elimination.
- We will not talk about loop unrolling which is another important optimization technique.

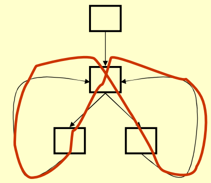
What is a Loop?

- Set of nodes
- Loop header
  - Single node
  - All iterations of loop go through header
- Back edge

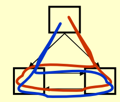


Anomalous Situations

- Two back edges, two loops, one header
- Compiler merges loops



- No loop header, no loop



Defining Loops With Dominators

Recall the concept of *dominators:*

- Node n dominates a node m if all paths from the start node to m go through n.
- The *immediate dominator* of m is the last dominator of m on any path from start node.
- A *dominator tree* is a tree rooted at the start node:
  - Nodes are nodes of control flow graph.
  - Edge from d to n if d is the immediate dominator of n.

Identifying Loops

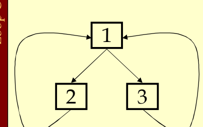
- A loop has a unique entry point – the header.
- At least one path back to header.
- Find edges whose heads (>) dominate tails (-), these edges are back edges of loops.
- Given a back edge n→d:
  - The node d is the loop header.
  - The loop consists of n plus all nodes that can reach n without going through d (all nodes "between" d and n)

Loop Construction Algorithm

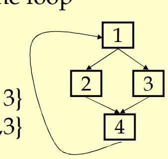
loop(d,n)
    loop = {d}; stack = ∅; *insert*(n);
    while stack not empty do
        m = pop stack;
        for all p ∈ *pred*(m) do *insert*(p);
insert(m)
    if m ∉ loop then
        loop = loop ∪ {m};
        push m onto stack;

Nested Loops

- If two loops do not have same header then
  - Either one loop (inner loop) is contained in the other (outer loop)
  - Or the two loops are disjoint
- If two loops have same header, typically they are unioned and treated as one loop

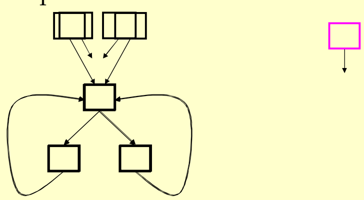


Two loops:
{1,2} and {1, 3}
Unioned: {1,2,3}



Loop Preheader

- Many optimizations stick code before loop.
- Put a special node (loop preheader) before loop to hold this code.



Loop Optimizations

- Now that we have the loop, we can optimize it!
- Loop invariant code motion:
  - Move loop invariant code to the header.

Loop Invariant Code Motion

If a computation produces the same value in every loop iteration, move it out of the loop.

```
for i = 1 to N
  x = x + 1
  for j = 1 to N
    a[i,j] = 100*N + 10*i + j + x
```

```
t1 = 100*N
for  i = 1 to N
   x = x + 1
   t2 = t1 + 10*i + x
   for  j = 1 to N
      a[i,j] =  t2 + j
```


Detecting Loop Invariant Code

- A statement is *loop-invariant* if operands are
  - Constant,
  - Have all reaching definitions outside loop, or
  - Have exactly one reaching definition, and that definition comes from an invariant statement
- Concept of exit node of loop
  - node with successors outside loop

Loop Invariant Code Detection Algorithm

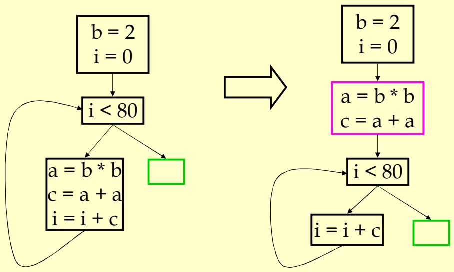
for all statements in loop
  if operands are constant or have all reaching definitions outside loop, mark statement as invariant
do
  for all statements in loop not already marked invariant
    if operands are constant, have all reaching definitions outside loop, or have exactly one reaching definition from invariant statement
      then mark statement as invariant
until there are no more invariant statements

Loop Invariant Code Motion

- Conditions for moving a statement s: x = y+z into loop header:
  - s dominates all exit nodes of loop
    - If it does not, some use after loop might get wrong value
    - Alternate condition: definition of x from s reaches no use outside loop (but moving s may increase run time)
  - No other statement in loop assigns to x
    - If one does, assignments might get reordered
  - No use of x in loop is reached by definition other than s
    - If one is, movement may change value read by use

Order of Statements in Preheader

Preserve data dependences from original program
(can use order in which discovered by algorithm)



Induction Variables

Example:

```
for j = 1 to 100
   *(&A + 4*j) = 202 - 2*j
```

Basic Induction variable:

J = 1, 2, 3, 4, …..

Induction variable &A+4*j:

&A+4*j = &A+4, &A+8, &A+12, &A+16, ….

What are induction variables?

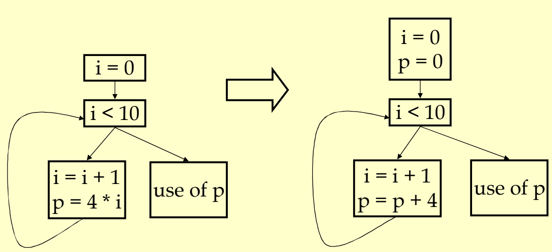
- x is an *induction variable* of a loop L if
  - variable changes its value every iteration of the loop
  - the value is a function of number of iterations of the loop
- In programs, this function is normally a linear function

  Example: for loop index variable j, function d + c*j

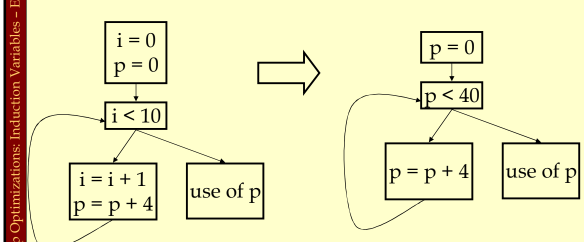
Types of Induction Variables

- Base induction variable:
  - Only assignments in loop are of form i = i ± c
- Derived induction variables:
  - Value is a linear function of a base induction variable.
  - Within loop, j = c*i + d, where i is a base induction variable.
  - Very common in array index expressions – an access to a[i] produces code like p = a + 4*i.

Strength Reduction for Derived Induction Variables



Elimination of Superfluous Induction Variables



Three Algorithms

- Detection of induction variables:
  - Find base induction variables.
  - Each base induction variable has a family of derived induction variables, each of which is a linear function of base induction variable.
- Strength reduction for derived induction variables.
- Elimination of superfluous induction variables.

Output of Induction Variable Detection Algorithm

- Set of induction variables:
  - base induction variables.
  - derived induction variables.
- For each induction variable j, a triple <i,c,d>:
  - i is a base induction variable.
  - the value of j is i*c+d.
  - j belongs to family of i.

Induction Variable Detection Algorithm

Scan loop to find all base induction variables

do

Scan loop to find all variables k with one assignment of form k = j*b where j is an induction variable with triple <i,c,d> (j = i*c + d, k = (i*c+d)*b = i*c*b + d*b)

make k an induction variable with triple <i,c*b,d*b>

Scan loop to find all variables k with one assignment of form k = j±b where j is an induction variable with triple <i,c,d> (j = i*c + d, k = (i*c+d) ± b = i*c*b + d ± b)

make k an induction variable with triple <i,c,b±d>

until no more induction variables are found

Strength Reduction

```
t = 202
for j = 1 to 100
  t = t - 2
  *(abase + 4*j) = t
```

Basic Induction variable:

J = 1, 2, 3, 4, …..

1 1 1

Induction variable 202 - 2*j

t = 202, 200, 198, 196, …..

-2 -2 -2

Induction variable abase+4*j:

abase+4*j = abase+4, abase+8, abase+12, abase+16, ….

4 4 4

Strength Reduction Algorithm

for all derived induction variables j with triple <i,c,d>

Create a new variable s

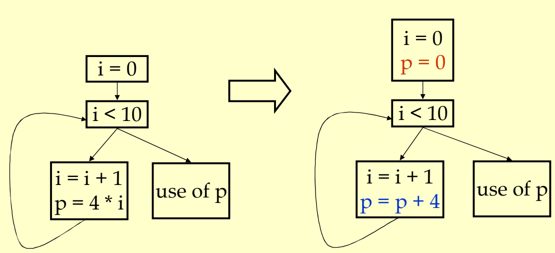
Replace assignment j = i*c+d with j = s

Immediately after each assignment i = i + e, insert statement s = s + c*e (c*e is constant)

place s in family of i with triple <i,c,d>

Insert s = c*i+d into preheader

Strength Reduction for Derived Induction Variables



Example

```
double A[256], B[256][256]
j = 1


while(j<100)
  A[j] = B[j][j]
  j = j + 2
```

```
double A[256], B[256][256]
j = 1
a = &A + 8
b = &B + 2056    // 2048+8
while(j<100)
  *a = *b
  j = j + 2
  a = a + 16
  b = b + 4112  // 4096+16
```


Induction Variable Elimination

Choose a base induction variable i such that
- only uses of i are in
  - termination condition of the form i < n
  - assignment of the form i = i + m

Choose a derived induction variable k with <i,c,d>
- Replace termination condition with k < c*n+d

Summary Loop Optimization

- Important because lots of time is spent in loops.
- Detecting loops.
- Loop invariant code motion.
- Induction variable analyses and optimizations:
  - Strength reduction.
  - Induction variable elimination.

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

- $r_i+r_j \to r_k$, 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

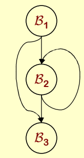
- LCM must insert copies to preserve redundant values
- LCM must compute its own map of expressions to unique ids

Lazy Code Motion: Running Example

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



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

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

Lazy Code Motion

Availability

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

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

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

Set **AVAILIN**(n_0) to ∅

Interpreting **AVAIL**

- 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 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

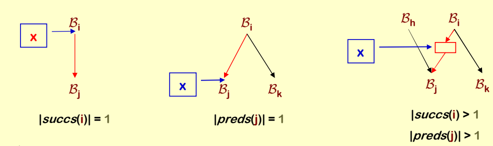
Compiler uses them to guide the rewrite step

- x ∈ INSERT(i,j) ⇒ insert x at start of i, end of j, or new block
- x ∈ 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

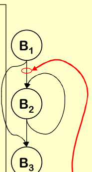
Edge placement
- x ∈ INSERT(i,j)



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

```
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	
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} }