Loop Optimizations

Loop Invariant Code Motion

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

```
t1 = 100*N
for i = 1 to N
  x = x + 1
  for j = 1 to N
     a(i,j) = 100*N + 10*i + j + x
```

→

- An expression can be moved out of the loop if all its operands are invariant in the loop

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

→

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

After Compiler Optimizations

- ex.: Copy Propagation, Algebraic Simplification.

Note:
CSE → Common Sub-Expression Elimination

Usefulness of LICM:

Reducing work inside a loop nest is very beneficial
- CSE of Expression ⇒ x instructions become x/2;
- LICM of Expression ⇒ x instructions become x/N.

Invariant Operands

- Constant Values
- Variables whose definitions are outside the loop
- Operand has only one reaching definition *and* that definition is loop invariant

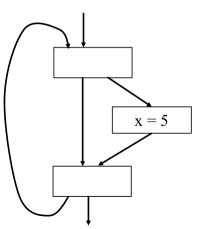
```
for i = 1 to N
  x = 100
  y = x * 5
```

```
for i = 1 to N
 if i > p then
   x = 10
 else
   x = 5
 y = x * 5
```

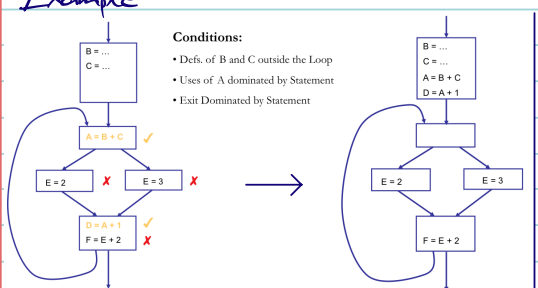
- Clearly a single definition is a safe restrictions
  - There could be many definition with the same value

Note

- Statement can be moved only if
  - All the Uses are Dominated by the Statement
  - The Exit of the Loop is Dominated by the Statement

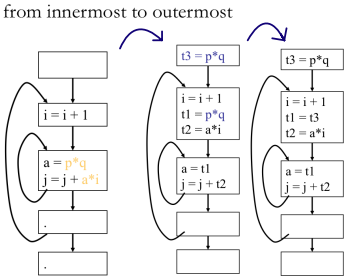


Example



Handling Nested Loops

- Process loops from innermost to outermost



Algorithm for LICM:

- Observations
  - Loop Invariant
    - Operands are defined outside loop or invariant themselves
  - Code Motion
    - Not all loop invariant instructions can be moved to pre-header.
    - Why?

- **Algorithm**
  - Find Invariant Expression
  - Check Conditions for Code Motion
  - Apply Code Transformation

Detecting Loop Invariant Computation

Algorithm

1. Compute Reaching Definitions for every variable in every Basic Block
2. Mark Invariant a statement `s: a = b+c` if
   - All definitions of `b` and `c` that reach the statement `s` are outside the loop
     - What about constants `b`, `c`?
3. Repeat: Mark Invariant if
   - All reaching definitions of `b` are outside the loop, or
   - There is exactly one reaching definition for `b`, and it is from a loop-invariant statement inside the loop
   - Idem for `c`
   - Until no changes to set of loop-invariant statements.

Code Motion Algorithm

- Given: a set of nodes in a loop
  - Compute Reaching Definitions
  - Compute Loop Invariant Computation
  - Compute Dominators
  - Find the exits of the loop, nodes with successors outside the loop
  - Candidate Statement for Code Motion:
    - Loop Invariant
    - In blocks that dominate all the Exits of the Loop
    - Assign to variable not assigned to elsewhere in the loop
    - In blocks that dominate all blocks in the loop that use the variable assigned
  - Perform a depth-first search of the blocks
    - Move candidate to pre-header if all the invariant operations it depends on have been moved

First Summary

- Loop Invariant Code Motion
  - Important and Profitable Transformation
  - Precise Definition and Algorithm for Loop Invariant computation
  - Precise Algorithm for code motion
- Combination of Several Analyses
  - Use of Reaching Definitions (DU-chains)
  - Use Dominators

Induction Variables in Loops

- What is an Induction Variable?
  - For a given loop variable v is an induction variable iff
    - Its value Changes at Every Iteration
    - Is either incremented or decremented by a Constant Amount
      - Either Compile-time Known or Symbolically Constant…
- Classification:
  - Basic Induction Variables
    - A single assignment in the loop of the form x = x + constant
    - Example: variable i in for i = 1 to 10
  - Derived Induction Variables
    - A linear function of a basic induction variable
    - variable j in the loop assigned $j = c_1 * i + c_2$

Importance

- Pervasive in Computations that Manipulate Arrays
  - Allow for Understanding of Data Access Patterns in Memory Access
    - Support Transformations Tailored to Memory Hierarchy
  - Can Be Eliminated with Strength Reduction
    - Substantially reduce the weight of address calculations
    - Combination with CSE
- Example:

```
for i = 1 to N
  for j = 1 to N
    a(i,j) = b(i,j)
```

```
for i = 1 to N
  t1 = @a(i,1)
  t2 = &b(i,1)
  for j = 1 to N
    *t1  = *t2
    t1 += 8
    t2 += 8
```

?

Detection of Induction Variables

- Algorithm:
  - Inputs: Loop L with Reaching Definitions and Loop Invariant
  - Output: For each Induction Variable j the triple (i,c,d) s.t. the value of j = i * c + d
  - Find the Basic Induction Variables by Scanning the Loop L such that each Basic Induction Variable has (i,1,0)
  - Search for variables k with a single assignment to k of the form:
    - k = j * b, k = b*j, k = j/b, k =+j with b a constant and j a basic induction variable
  - Check if the assignment dominates the definition points for j

Example between the slides 50 and 57.

Second Summary

- Induction Variables
  - Change Values at Every Iteration of a Loop by a Constant amount
  - Basic and Derived Induction Variables with Affine Relation
- Great Opportunity for Transformations
  - Pervasive in Loops that Manipulation Array Variables
  - Loop Control and Array Indexing
- Combination of Various Analyses and Transformations
  - Dominators, Reaching Definitions
  - Strength Reduction, Dead Code Elimination and Copy Propagation and Common Sub-Expression Elimination