Loop Optimizations

Loop Invariant Code Motion
Induction Variables
Outline

• Loop Invariant Code Motion

• Induction Variables Recognition

• Combination of Analyses
Loop Invariant Code Motion

• If a computation produces the same value in every loop iteration, move it out of the loop
Loop Invariant Code Motion

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

```plaintext
for i = 1 to N
    x = x + 1
for j = 1 to N
    a(i,j) = 100*N + 10*i + j + x
```
Loop Invariant Code Motion

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

```plaintext
for i = 1 to N
    x = x + 1
for j = 1 to N
    a(i,j) = 100*N + 10*i + j + x
```
Loop Invariant Code Motion

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

\[ t1 = 100*N \]

\[
\text{for } i = 1 \text{ to } N \\
\quad x = x + 1 \\
\text{for } j = 1 \text{ to } N \\
\quad a(i,j) = 100*N + 10*i + j + x
\]
Loop Invariant Code Motion

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

\[ t1 = 100 \times N \]

\[
\text{for } i = 1 \text{ to } N \\
\quad \text{x = x + 1} \\
\quad \text{for } j = 1 \text{ to } N \\
\quad \quad a(i,j) = t1 + 10 \times i + j + x
\]
Loop Invariant Code Motion

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

\[ t_1 = 100 \times N \]

\[ \text{for } i = 1 \text{ to } N \]

\[ x = x + 1 \]

\[ \text{for } j = 1 \text{ to } N \]

\[ a(i,j) = t_1 + 10 \times i + j + x \]
Loop Invariant Code Motion

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

\[ t1 = 100 \times N \]

\[ \text{for } i = 1 \text{ to } N \]

\[ x = x + 1 \]

\[ \text{for } j = 1 \text{ to } N \]

\[ a(i,j) = t1 + 10 \times i + j + x \]
Loop Invariant Code Motion

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

\[ t1 = 100 \times N \]

\[ \text{for } i = 1 \text{ to } N \]

\[ x = x + 1 \]

\[ t2 = 10 \times i + x \]

\[ \text{for } j = 1 \text{ to } N \]

\[ a(i,j) = t1 + 10 \times i + j + x \]
Loop Invariant Code Motion

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

\[ t_1 = 100*N \]

\[ \text{for } i = 1 \text{ to } N \]
\[ x = x + 1 \]

\[ t_2 = 10*i + x \]

\[ \text{for } j = 1 \text{ to } N \]
\[ a(i,j) = t_1 + t_2 + j \]
Loop Invariant Code Motion

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

```plaintext
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
```
Loop Invariant Code Motion

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

\[ t_1 = 100 \times N \]

\[
\text{for } i = 1 \text{ to } N \\
\quad x = x + 1 \\
\quad t_2 = 10 \times i + x \\
\text{for } j = 1 \text{ to } N \\
\quad a(i,j) = t_1 + t_2 + j
\]

• Correctness and Profitability?
  – Loop Should Execute at Least Once!
Opportunities for Loop Invariant Code Motion

• In User Code
  – Complex Expressions
  – Easily readable code, reduce # of variables

• After Compiler Optimizations
  – Copy Propagation, Algebraic simplification
Usefulness of Loop Invariant Code Motion

• In many programs most of the execution happens in loops

• 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
Implementing Loop Invariant Code Motion

• If a computation produces the same value in every loop iteration, move it out of the loop
• An expression can be moved out of the loop if all its operands are invariant in the loop
Invariant Operands

• Constants are invariant

DUH!!!
Invariant Operands

• Constants are invariant
• All the definitions are outside the loop
Invariant Operands

• All the definitions are outside the loop

\[ x = f(...) \]
\[ y = g(...) \]
\[ \text{for } i = 1 \text{ to } N \]
\[ z = z + x*y \]
Invariant Operands

• All the definitions are outside the loop

\[
\begin{align*}
x &= f(...) \\
y &= g(...) \\
t &= x \times y \\
\text{for } i &= 1 \text{ to } N \\
\quad z &= z + t
\end{align*}
\]
Invariant Operands

• Operand has only one reaching definition and that definition is loop invariant

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

• Operand has only \textit{one} reaching definition \textit{and} that
definition is loop invariant

\begin{align*}
\text{for } i = 1 \text{ to } N & \quad \text{for } i = 1 \text{ to } N \\
x &= 100 & x &= 100 \\
y &= x \times 5 & y &= x \times 5
\end{align*}
Invariant Operands

- Operand has only one reaching definition and that definition is loop invariant

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

Definition outside the loop, in a basic blocks that dominates the loop header…
Invariant Operands

• Operand has only one reaching definition and that definition is loop invariant.

\[
\begin{align*}
\text{for } i = 1 \text{ to } N & \quad \text{for } i = 1 \text{ to } N \\
x & = 100 \\
y & = x \times 5 \\
y & = 100 \times 5
\end{align*}
\]
Invariant Operands

- Operand has only *one* reaching definition *and* that definition is loop invariant

```plaintext
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 restriction
  - There could be many definition with the same value
Move Or Not To Move….

- Statement can be moved only if
  - All the Uses are Dominated by the Statement

\[ x = 5 \]

\[ i = i + x \]
Move Or Not To Move…. 

• Statement can be moved only if
  – All the Uses are Dominated by the Statement

```plaintext
x = f(i)

x = 5

i = i + x
```
Move Or Not To Move….

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

  – **Reaching Definitions (RD) Analysis** computes all the definitions of \( x \) and \( y \) which may reach \( t = x \text{ OP } y \)
Conditions for Code Motion

• Correctness: Movement doesn’t change *semantics* of the program
• Performance: Code is not slowed down

• Basic Ideas: Defines **once** and **for all**
  – Control flow
  – Other definitions
  – Other uses

\[
A = B + C
\]
Example: Loop Invariant Code Motion

\[ \begin{align*}
E &= 2 \\
E &= 3 \\
D &= A + 1 \\
F &= E + 2 \\
A &= B + C \\
B &= \ldots \\
C &= \ldots
\end{align*} \]
Example: Loop Invariant Code Motion

Conditions:
- Defs. of B and C outside the Loop
- Uses of A dominated by Statement
- Exit Dominated by Statement
Example: Loop Invariant Code Motion

Conditions:
- Defs. of B and C outside the Loop
- Uses of A dominated by Statement
- Exit Dominated by Statement

```plaintext
B = ...
C = ...

A = B + C

E = 2
E = 3

D = A + 1
F = E + 2
```
Example: Loop Invariant Code Motion

Conditions:
- Defs. of B and C outside the Loop
- Uses of A dominated by Statement
- Exit Dominated by Statement

D = A + 1
F = E + 2

E = 2
E = 3

B = ...
C = ...

A = B + C

\[ E = 2 \]
\[ E = 3 \]
Example: Loop Invariant Code Motion

Conditions:

- Defs. of B and C outside the Loop
- Uses of A dominated by Statement
- Exit Dominated by Statement

B = ...
C = ...
A = B + C
D = A + 1

E = 2

E = 3

F = E + 2
Other Issues

• **Preserve dependencies** between loop-invariant instructions when hoisting code out of the loop

```c
for (i=0; i<N; i++) {
    x = y+z;
    t = x*x;
    a[i] = 10*i + x*x;
}
```

• **Nested loops**: apply loop invariant code motion algorithm multiple times

```c
for (i=0; i<N; i++)
    for (j=0; j<M; j++)
        a[i][j] = x*x + 10*i + 100*j;
```

```c
for (i=0; i<N; i++)
    t1 = x*x;
    for (i=0; i<N; i++)
        t2 = t1 + 10*i;
        for (j=0; j<M; j++)
            a[i][j] = t2 + 100*j;
```
Handling Nested Loops

• Process Loops from Innermost to Outermost
Handling Nested Loops

- Process Loops from Innermost to Outermost

```
\[
\begin{align*}
  i &= i + 1 \\
  a &= p \times q \\
  j &= j + a \times i \\
  &\ldots
\end{align*}
\]
```
Handling Nested Loops

• Process Loops from Innermost to Outermost

\[
a = p \times q \\
j = j + a \times i \\
i = i + 1
\]
Handling Nested Loops

• Process Loops from Innermost to Outermost

```
i = i + 1
j = j + a*i
a = p*q
j = j + a*i
. . .
```

```
i = i + 1
j = j + a*i
. . .
```
Handling Nested Loops

• Process Loops from Innermost to Outermost
Handling Nested Loops

• Process Loops from Innermost to Outermost

```
a = t1
j = j + t2
i = i + 1
t1 = p*q
t2 = a*i
```
Handling Nested Loops

• Process Loops from Innermost to Outermost

```
a = t1
j = j + t2
i = i + 1
t1 = p*q
t2 = a*i
```
Handling Nested Loops

- Process Loops from Innermost to Outermost

```plaintext
a = t1
j = j + t2

i = i + 1
t1 = p*q

```
Handling Nested Loops

• Process Loops from Innermost to Outermost

\[
\begin{align*}
\text{t3} &= p \times q \\
\text{i} &= \text{i} + 1 \\
\text{t1} &= \text{t3} \\
\text{t2} &= \text{a} \times \text{i} \\
\text{a} &= \text{t1} \\
\text{j} &= \text{j} + \text{t2}
\end{align*}
\]
Handling Nested Loops

- Process Loops from Innermost to Outermost

\[
\begin{align*}
t3 &= p \times q \\
i &= i + 1 \\
t1 &= t3 \\
t2 &= a \times i \\
a &= t1 \\
j &= j + t2
\end{align*}
\]
Algorithm for Loop Invariant Code Motion

• 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

Input: Basic Blocks and CFG, Dominator Relation, Loop Information

Output: Instructions that are Loop Invariant

\[ \text{InvSet} = \emptyset \]

repeat

for each instruction \( i \notin \text{InvSet} \)

if operands are constants, or

have definitions outside the loop, or

have exactly one definition \( d \in \text{InvSet} \);

then

\[ \text{InvSet} = \text{InvSet} \cup \{i\} \];

until no changes in InvSet;
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
More Examples

D = A + 1
A = B + C
E = 3
E = 2
F = E + 2
A = B + C
D = A + 1
E = 3
More Aggressive Optimizations

• Gamble On: Most loops get executed
  – Can we relax the constraint of dominating all exits?

  ```
  while p do 
    s
  if p {
    pre-header
    repeat
    statements
    until not p;
  }
  ```

• Landing Pads
Outline

• 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
  – Use Dominators

• Next: Combination with Loop Induction Variables
Redundancy Elimination

• Two “Optimizations”
  – Common Sub-Expression Elimination (CSE)
  – Loop Invariant Code Motion (LICM)
  – Dead Code Elimination

• Many Others
  – Value Numbering
  – Partial Redundancy Elimination (PRE)

• Focus: Induction Variable Recognition & Elimination
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 \( x = x + \text{constant} \)
    • 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 \( j = c_1 \times i + c_2 \)
Why Are Induction Variables Important?

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

  ```c
  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 k the triple \((i,c,d)\) s.t. the value of \(k = i \times c + d\)
    
    – 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:
      • \(k = i \times b, k = b \times i, k = i/b\) with b a constant and i is a Basic Induction Variable
    
    – Check if the Assignment to k is dominated by the definitions for i
    – If so, k is a Derived Induction Variable of i
Strength Reduction & Induction Variables

• Idea of the Transformation
  – Replace the Induction Variable in each Family by references to a common induction variable, the basic induction variable.
  – Exploit the Algebraic Properties for the update to the basic variable

• Algorithm outline
  foreach Basic Induction variable i do
    foreach Induction variable k: (i,c,d) in the family of i do
      create a new variable s
      replace the assignment to k by k = s
      after each assignment i = i + n where n is a constant
        append s = s + c * n
      place s in the family of induction variables of i
    end foreach
    initialize s to c*i + d on loop entry as
      either s = c * i followed by s = s + d (simplify if d = 0 or c = 1)
  end foreach
Detection of Induction Variables Example

\[
\begin{align*}
i & = m - 1 \quad \text{B1} \\
j & = n \\
t_1 & = 4 \times n \\
v & = a[t_1] \\
\end{align*}
\]

\[
\begin{align*}
i & = i + 1 \quad \text{B2} \\
t_2 & = 4 \times i \\
t_3 & = a[t_2] \\
\text{if } t_3 & < v \text{ goto B2} \\
\end{align*}
\]

\[
\begin{align*}
j & = j - 1 \quad \text{B3} \\
t_4 & = 4 \times j \\
t_5 & = a[t_4] \\
\text{if } t_5 & > v \text{ goto B3} \\
\end{align*}
\]

\[
\text{if } i < j \text{ goto B2} \quad \text{B4}
\]
Detection of Induction Variables Example

- Basic Induction Variables:
  - \( i \) in B2: single increment, \((i,1,1)\)
  - \( j \) in B3: single decrement \((j,1,-1)\)
Detection of Induction Variables Example

- **Basic Induction Variables:**
  - i in B2: single increment, (i,1,1)
  - j in B3: single decrement (j,1,-1)

- **Derived Induction Variables**
  - t2 in B2: single assign (i,4,0)
  - t4 in B3: single assign (j,4,0)
Strength Reduction of Induction Variables

- **Basic Induction Variables:**
  - i in B2: single increment, (i,1,1)
  - j in B3: single decrement (j,1,-1)

- **Derived Induction Variables**
  - t2 in B2: single assign (i,4,0)
  - t4 in B3: single assign (j,4,0)

- **Strength Reduction (for t4 in B3)**
  - create s4 for the expression 4*j
  - replace t4 = 4*j with t4 = s4
  - replace induction step for j with s4 = s4 - 4 where -4 comes from -1*c
  - create initialization of s4 in pre-header
Eliminating Induction Variables

• After all the “tricks” we might be left with
  – Code that uses the basic induction variables just for conditional
    including the loop control

• Given the linear relation between induction variables
  – we can remove the basic induction variable by rewording the tests with
    a derived induction variable that is used in the code.
  – Check out dead statements (trivial is you use SSA)
  – Check the initialization and remove induction variables.
Strength Reduction of Induction Variables

- Basic Induction Variables:
  - $i$ in B2: single increment, $(i, 1, 1)$
  - $j$ in B3: single decrement $(j, 1, -1)$

- Derived Induction Variables
  - $t_2$ in B2: single assign $(i, 4, 0)$
  - $t_4$ in B3: single assign $(j, 4, 0)$

- Strength Reduction (for $t_4$ in B3)
  - create $s_4$ for the expression $4 \cdot j$
  - replace $t_4 = 4 \cdot j$ with $t_4 = s_4$
  - replace induction step for $j$ with $s_4 = s_4 - 4$ where $-4$ comes from $-1 \ast c$
  - create initialization of $s_4$ in pre-header

- Elimination of Induction Variables
  - replace $i < j$ with $s_2 < s_4$
  - copy propagate $s_2$ and $s_4$
**Strength Reduction of Induction Variables**

- **Basic Induction Variables:**
  - i in B2: single increment, (i,1,1)
  - j in B3: single decrement (j,1,-1)

- **Derived Induction Variables**
  - t2 in B2: single assign (i,4,0)
  - t4 in B3: single assign (j,4,0)

- **Strength Reduction (for t4 in B3)**
  - create s4 for the expression 4*j
  - replace t4 = 4*j with t4 = s4
  - replace induction step for j with s4 = s4 - 4 where -4 comes from -1*c
  - create initialization of s4 in pre-header

- **Elimination of Induction Variables**
  - replace i < j with s2 < s4
  - copy propagate s2 and s4

```plaintext
B1
\[ t1 = 4 \times n \]
\[ v = a[t1] \]
\[ s2 = 4 \times (m-1) \]
\[ s4 = 4 \times n \]

B2
\[ s2 = s2 + 4 \]
\[ t2 = s2 \]
\[ t3 = a[s2] \]
\[ \text{if } t3 < v \text{ goto B2} \]

B3
\[ s4 = s4 - 4 \]
\[ t4 = s4 \]
\[ t5 = a[s4] \]
\[ \text{if } t5 > v \text{ goto B3} \]

B4
\[ \text{if } s2 < s4 \text{ goto B2} \]
```
Summary

• Loop Invariant Code Motion
  – Important and Profitable Transformation
  – Precise Definition and Algorithm for Loop Invariant Computation
  – Precise Algorithm for Code Motion

• Induction Variables
  – Change Values at Every Iteration of a Loop by a Constant amount
  – Basic and Derived Induction Variables with Affine Relation

• Combination of Various Analyses and Transformations
  – Dominators, Reaching Definitions
  – Strength Reduction, Dead Code Elimination and Copy Propagation and
    Common Sub-Expression Elimination