

CS406: Compilers

Spring 2022

Week 11: Loop Optimization, ..

Short Quiz

- <https://forms.gle/Vz5p3aTsTk5RFuss7>

Optimize Loops

- Example - Code Motion

Should be careful while doing optimization of loops

```
while J > I loop  
    A(j) := 10/I;  
    j := j + 2;  
end loop;
```

Optimize Loops – Code Motion

- Should be careful while doing optimization of loops

```
while J > I loop  
    A(j) := 10/I;  
    j := j + 2;  
end loop;
```

- Optimization: can move $10/I$ out of loop.

Optimize Loops – Code Motion

- Should be careful while doing optimization of loops

```
while J > I loop  
    A(j) := 10/I;  
    j := j + 2;  
end loop;
```

- Optimization: can move $10/I$ out of loop
- What if $I = 0$?

Optimize Loops – Code Motion

- Should be careful while doing optimization of loops

```
while J > I loop  
    A(j) := 10/I;  
    j := j + 2;  
end loop;
```

- Optimization: can move $10/I$ out of loop
- What if $I = 0$?
- What if $I \neq 0$ but loop executes zero times?

Optimization Criteria - Safety and Profitability

- **Safety** - is the code produced after optimization producing same result?
- **Profitability** - is the code produced after optimization running faster or uses less memory or triggers lesser number of page faults etc.

```
while J > I loop  
    A(j) := 10/I;  
    j := j + 2;  
end loop;
```

- E.g. moving I out of the loop introduces exception (when I=0)
- E.g. if the loop is executed zero times, moving $A(j) := 10/I$ out is not profitable

Optimize Loops – Code Generation

- The outline of code generation for ‘for’ loops looked like this:

```
for (<init_stmt>;<bool_expr>;<incr_stmt>)  
  <stmt_list>  
end
```

↓

```
<init_stmt>  
LOOP:  
  <bool_expr>  
  j<!op> OUT  
  <stmt_list>  
INCR:  
  <incr_stmt>  
  jmp LOOP  
OUT:
```

```
for (i=0; i<=255;i++) {  
  <stmt_list>  
}
```

↓ **Naïve code generation**

```
code for i=0;  
LOOP:  code for i<=255  
        jump0 OUT  
        code for <stmt_list>  
INCR:  code for i++  
        jump LOOP  
OUT:
```

Question: why naïve is not good?

Optimize Loops – Code Generation

- What happens when ub is set to the maximum possible integer representable by the type of i?

```
for (i=0; i<=255;i++) {  
    <stmt_list>  
}
```

Better code:



```
code for i=0;  
code for lb=1, ub=255  
code for lb<=ub  
jump0 OUT
```

```
LOOP:  code for <stmt_list>  
       code for lb=ub  
       jump1 OUT
```

```
INCR:  code for i++  
       jump LOOP
```

```
OUT:
```

—————→
generalizing:

```
code for i=0;  
compute lb, ub  
code for lb<=ub  
jump0 OUT  
assign index=lb  
assign limit=ub
```

```
LOOP:  code for <stmt_list>  
       code for index=limit  
       jump1 OUT
```

```
INCR:  code for increment index  
       jump LOOP
```

```
OUT:
```

Optimize Loops -Identifying Invariant Expressions

- How do we identify expressions that can be moved out of the loop?
 - LoopDef = { } set of variables defined (i.e. whose values are overwritten) in the loop body
 - LoopUse = { } 'relevant' variables used in computing an expression

```
Mark_Invariants(Loop L) {  
    1. Compute LoopDef for L  
    2. Mark as invariant all expressions,  
       whose relevant variables don't belong  
       to LoopDef  
}
```

Optimize Loops -Identifying Invariant Expressions

- Example

LoopDef{}

```
for I = 1 to 100      → {A, J, K, I}
  for J = 1 to 100    → {A, J, K}
    for K = 1 to 100  → {A, K}
      A[I][J][K] = (I*J)*K
```

Optimize Loops -Identifying Invariant Expressions

- Example

Invariant
Expressions

```
for I = 1 to 100
  for J = 1 to 100
    for K = 1 to 100 → { I*J,
      A[I][J][K] = (I*J)*K      Addr(A[i][j]) }
```

For an array access, $A[m] \Rightarrow \text{Addr}(A) + m$

For 3D array above*, $\text{Addr}(A[I][J][K]) =$

$$\text{Addr}(A) + (I*10000) - 10000 + (J*100) - 100 + K - 1$$

*Assuming row-major ordering of storage

Optimize Loops -Identifying Invariant Expressions

- Example

Invariant
Expressions

```
for I = 1 to 100
  for J = 1 to 100
    for K = 1 to 100
      A[I][J][K] = (I*J)*K
```

→ { Addr(A[i]) }

For an array access, $A[m] \Rightarrow \text{Addr}(A) + m$

For 3D array above*, $\text{Addr}(A[I][J][K]) =$

$$\text{Addr}(A) + (I*10000) - 10000 + (J*100) - 100 + K - 1$$

Optimize Loops -Factoring Invariant Expressions

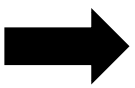
- Move the invariant expressions identified

```
Factor_Invariants(Loop L) {  
    Mark_Invariants(L);  
    foreach expression E marked an invariant:  
        1. Create a temporary T  
        2. Replace each occurrence of E in L with T  
        3. Insert T:=E in L's header code  
           //If loop is known to execute at least once,  
           insert T:=E before LOOP:  
}
```

Optimize Loops -Factoring Invariant Expressions

- Example

```
for I = 1 to 100
  for J = 1 to 100
    for K = 1 to 100
      A[I][J][K] = (I*J)*K
```



```
for I=1 to 100
  temp3=Addr(A[i])
  for J=1 to 100
    temp1=Addr(temp3(J))
    temp2=I*J
    for K=1 to 100
      temp1[K]=temp2*K
```

Optimize Loops -Factoring Invariant Expressions

- Expressions cannot always be moved out!

Case I: We can move $t = a \text{ op } b$ if the statement dominates all loop exits where t is live

A node a dominates node b if all paths to b must go through a

```
for (...) {  
    if(*)  
        a = 100  
}  
c=a
```

Cannot move $a=100$ because it does not dominate $c=a$ i.e. there is one path (when if condition is false) $c=a$ can be reached without going $a=100$

Optimize Loops -Factoring Invariant Expressions

- Expressions cannot always be moved out!

Case II: We can move $t = a \text{ op } b$ if there is only one definition of t in the loop

```
for (...) {  
    if(*)  
        a = 100  
    else  
        a = 200  
}
```

Multiple definition of a

Optimize Loops -Factoring Invariant Expressions

- Expressions cannot always be moved out!

Case III: We can move $t = a \text{ op } b$ if t is not defined before the loop, where the definition reaches t 's use after the loop

```
a=5
for (...) {
    a = 4+b
}
c=a
```

Definition of a in $a=5$ reaches $c=a$, which is defined after the loop

Optimize Loops –Strength Reduction

- Like strength reduction in peephole optimization
 - E.g. replace $a*2$ with $a<<1$
- Applies to uses of **induction variable** in loops
 - **Basic induction variable (I)** – only definition within the loop is of the form $I = I \pm S$, (S is loop invariant)
I usually determines number of iterations
 - **Mutual induction variable (J)** – defined within the loop, its value is linear function of other induction variable, I, such that
$$J = I * C \pm D \quad (C, D \text{ are loop invariants})$$

Optimize Loops –Strength Reduction

```
strength_reduce(Loop L) {  
    Mark_Invariants(L);  
    foreach expression  $E$  of the form  $I * C + D$  where  $I$  is  
    L's loop index and  $C$  and  $D$  are loop invariants  
        1. Create a temporary  $T$   
        2. Replace each occurrence of  $E$  in  $L$  with  $T$   
        3. Insert  $T := I_0 * C + D$ , where  $I_0$  is the initial value of the  
           induction variable, immediately before  $L$   
        4. Insert  $T := T + S * C$ , where  $S$  is the step size, at the end of  
           L's body  
}
```

Optimize Loops –Strength Reduction

- Suppose induction variable I takes on values I_0 , I_0+S , I_0+2S , $I_0+3S \dots$ in iterations 1, 2, 3, 4, and so on...
- Then, in consecutive iterations, Expression $I*C+D$ takes on values

$$I_0 * C + D$$

$$(I_0 + S) * C + D = I_0 * C + S * C + D$$

$$(I_0 + 2S) * C + D = I_0 * C + 2S * C + D$$

- The expression \ddots changes by a constant $S * C$
- Therefore, we have replaced a $*$ and $+$ with a $+$

Optimize Loops – Strength Reduction

- Example (Applying to innermost loop)

```
for I = 1 to 100
  for J = 1 to 100
    for K = 1 to 100
      A[I][J][K] = (I*J)*K
      . . .
      temp2=I*J
      temp4=temp2
      for K=1 to 100
        temp1[K]=temp4
        temp4=temp4+temp2
      //S=1
      //C=temp2
```

→

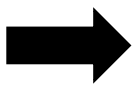
```
for I=1 to 100
  temp3=Addr(A[i])
  for J=1 to 100
    temp1=Addr(temp3(J))
    temp2=I*J
    for K=1 to 100
      temp1[K]=temp2*K
    ↙
```

```
for K=1 to 100
  temp1[K]=temp4
  temp4=temp4+temp2
```


Optimize Loops – Strength Reduction

- Exercise (Apply to intermediate loop)

```
for I=1 to 100
  temp3=Addr(A[i])
  for J=1 to 100
    temp1=Addr(temp3(J))
    temp2=I*J
    for K=1 to 100
      temp1[K]=temp2*K
```



```
    . . .
    temp2=I*J
    temp4=temp2
    for K=1 to 100
      temp1[K]=temp4
      temp4=temp4+temp2
```



```
// Induction var = J
// S = 1
// Expression = I * J
```

Optimize Loops – Strength Reduction

- Exercise (Apply to intermediate loop)

...
temp5=I

for J=1 to 100

temp1=Addr(temp3(J))

temp2=temp5

temp4=temp2

for K=1 to 100

temp1[K]=temp4

temp4=temp4+temp2

temp5=temp5+I

... → ...



Optimize Loops – Strength Reduction

- Further strength reduction possible?

```
for I=1 to 100
    temp3=Addr(A[i])
    temp5=I
    for J=1 to 100
        temp1=Addr(temp3(J))
        temp2=temp5
        temp4=temp2
        for K=1 to 100
            temp1[K]=temp4
            temp4=temp4+temp2
        temp5=temp5+I
```

Optimize Loops – Loop Unrolling

- Modifying induction variable in each iteration can be expensive
- Can instead *unroll* loops and perform multiple iterations for each increment of the induction variable
- What are the advantages and disadvantages?

```
for (i = 0; i < N; i++)  
    A[i] = ...
```



Unroll by factor of 4

```
for (i = 0; i < N; i += 4)  
    A[i] = ...  
    A[i+1] = ...  
    A[i+2] = ...  
    A[i+3] = ...
```

Optimize Loops - Summary

- Low level optimization
 - Moving code around in a single loop
 - Examples: loop invariant code motion, strength reduction, loop unrolling
- High level optimization
 - Restructuring loops, often affects multiple loops
 - Examples: loop fusion, loop interchange, loop tiling