Preliminary Transformations

Auxiliary Induction Variable Substitution and Loop Normalization

Overview

- Goal: improve accuracy of dependence testing
  - Conventional testing methods assume a closed form of
    - loop index variables (aka, loop induction variables)
    - loop invariants (which can be treated as constants)
- Transformations to put more subscripts into standard form
  - Induction Variable Substitution: remove unknown variables
  - Loop Normalization: testing is easier if loop strides are 1
  - Related optimizations
    - redundancy elimination, dead code elimination, constant propagation
    - Help find more loop invariant expressions
- Optimizations by programmers often confuse compilers
  - Leave optimizations to compilers?

```
INC=2  KI = 0
DO I = 1, 100
   DO J = 1, 100
      KI = KI + INC
      U(KI) = U(KI) + W(J)
   ENDDO
   S(I) = U(KI)
ENDDO
```

Example: Auxiliary Induction Variable Substitution

Original code

```
INC=2  KI = 0
DO I = 1, 100
   DO J = 1, 100
       KI = KI + INC
       U(KI) = U(KI) + W(J)
   ENDDO
   S(I) = U(KI)
ENDDO
```

KI is a function of loop index variable J

```
INC = 2     KI = 0
DO I = 1, 100
   DO J = 1, 100
      ! Deleted: KI = KI + INC
        U(KI + J*INC) = U(KI + J*INC) + W(J)
   ENDDO
   KI = KI + 100 * INC
   S(I) = U(KI)
ENDDO
```

KI is a function of loop index variable I

```
INC = 2   KI = 0
DO I = 1, 100
  DO J = 1, 100
        U(KI + (I-1)*100*INC + J*INC) =
                U(KI + (I-1)*100*INC + J*INC) + W(J)
  ENDDO
  ! Deleted: KI = KI + 100 * INC
  S(I) = U(KI + I * (100*INC) )
ENDDO
KI = KI + 100 * 100 * INC
```

Now KI is loop invariant (no longer modified inside the loops)

Example: Constant Propagation

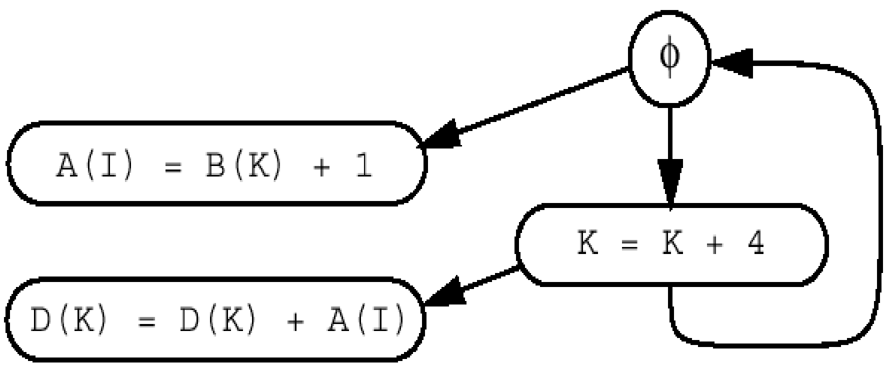
```
INC = 2   KI = 0
DO I = 1, 100
  DO J = 1, 100
        U(KI + (I-1)*100*INC + J*INC) =
                U(KI + (I-1)*100*INC + J*INC) + W(J)
  ENDDO
  ! Deleted: KI = KI + 100 * INC
  S(I) = U(KI + I * (100*INC) )
ENDDO
KI = KI + 100 * 100 * INC
```

```
INC = 2
! Deleted: KI = 0
DO I = 1, 100
  DO J = 1, 100
        U(I*200 + J*2 - 200) =
                U(I*200 + J*2 -200) + W(J)
  ENDDO
  S(I) = U(I*200)
ENDDO
KI = 20000
```

Induction Variable Substitution

- Definition: auxiliary induction variable
  - Any variable that can be expressed as cexpr * I + iexpr everywhere it is used in a loop, where
    - I is the loop index variable
    - cexpr and iexpr are loop-invariant expressions (their values do not vary in the loop)
  - Different locations in the loop may require substitution of different values of iexpr
- Example:

```
DO I = 1, N
   A(I) = B(K) + 1
   K = K + 4
   …
   D(K) = D(K) + A(I)
ENDDO
```



Induction Variable Substitution

- Recognizing auxiliary induction variables
  - Use data-flow analysis to build def-use chain or SSA representation
    - Connect each variable use with possible definitions that produce its value
    - Connect each variable definition with possible uses of the produced value
    - The algorithm in the textbook uses SSA
  - For each loop L, recognize loop invariant variables and expressions
    - Variables and expressions whose values never change inside L
  - For each loop L, Recognize auxiliary induction variables
    - Variables modified at each iteration of L by incrementing/decrementing it with a loop invariant value
- Substitute auxiliary induction variables
  - For each loop L:do I=L,U,S from inside out and each AIV iv of L
    - Let s: iv=iv+cexpr be the statement that modifies iv inside L
    - For each expression exp in L that uses iv before s:
      replace exp with exp+(I-L)/S*cexpr
    - For each expression exp in L that uses iv after s:
      replace exp with exp+(I-L+S)/S*cexpr
    - Delete s and modify def-use chain/SSA accordingly
    - If iv is used after loop L : insert iv=iv+ (U-L+S)/S * cexpr after loop L

Are We Missing Something?

- More complex example

```
DO I = 1, N, 2
   K = K + 1
   A(K) = A(K) + 1
   K = K + 1
   A(K) = A(K) + 1
ENDDO
```

- Solution: forward substitute the use of a variable v in stmt S if
  - There is a single definition def(v) that can reach S
  - The value assigned to v does not change between def(v) and S
  - If RHS of def(v) includes v, need to remove def(v) after substitution

```
DO I = 1, N, 2
   A(K+1) = A(K+1) + 1
   K = K+1 + 1
   A(K) = A(K) + 1
ENDDO
```

Forward Expression Substitution

```
DO I = 1, 100
  K = I + 2
  A(K) = A(K) + 5
ENDDO
```

→

```
DO I = 1, 100
  A(I+2) = A(I+2) + 5
ENDDO
```

- Need definition-use edges and control flow analysis
- Need to guarantee
  - The definition does not have unknown side-effect (e.g.,I/O)
  - The definition is always evaluated before the use (i.e., it is the only def that can reach the use)
  - The RHS of definition does not change before the uses
    - Approximation: RHS includes only loop index variables and loop invariants
- Would like to substitute a definition S only if it is in loop L
  - Test whether level-K loop containing S is equal to L
- Modify the definition: reorder def and uses if necessary
  - If substitution has been applied to all uses: remove the definition
  - If substitution has been applied to all uses inside loop: move definition outside of the loop

Induction Variable Substitution

```
procedure IVDrive(L);
    // L is the loop being processed, assume SSA graph available
    // IVDrive performs forward substitution and induction variable
    // substitution on the loop L, recursively calling itself where
    // necessary.

    foreach statement S in L in order do
        case(kind(S))
            assignment:
                FS_not_done := ForwardSub(S,L);
                if FS_not_done then IVSub(S,L);
            DO-loop:
                IVDrive(S);
            default:
        end case
    end do
end IVDrive;
```

Loop Normalization

- Goal: modify loops to have lower bound 1 with stride 1
  - To make dependence testing as simple as possible
  - Serves as information gathering phase
- Algorithm for normalizing a loop L0: do I=L,U,S
  - i = a unique compiler-generated LIV
  - Replace the loop header for L0 with

    do i = 1, (U – L + S) / S, 1
  - Replace each reference to I within the loop by

    i * S – S + L;
  - insert a finalization assignment I = i * S – S + L; immediately after the end of the loop

Tradeoff of Applying Loop Normalization

Un-normalized:

```
DO I = 1, M
  DO J = I, N
    A(J, I) = A(J, I - 1) + 5
  ENDDO
ENDDO
```

Has a direction vector of (<,=)

Normalized:

```
DO I = 1, M
  DO J = 1, N – I + 1
    A(J + I – 1, I) =
        A(J + I – 1, I – 1) + 5
  ENDDO
ENDDO
```

Has a direction vector of (<,>)

- Consider interchanging loops
  - (<,=) becomes (=,>) OK
  - (<,>) becomes (>,<) Problem
- What if the step size is symbolic?
  - Prohibits dependence testing
    - Workaround: use step size 1
    - Less precise, but allow dependence testing

IV Substitution and Loop Normalization

- IVSub without loop normalization
  - Problem: inefficient code; nonlinear subscript

```
DO I = L, U, S
   K = K + N
   … = A(K)
ENDDO
```

⇒

```
DO I = L, U, S
   … = A(K + (I – L + S) / S * N)
ENDDO
K = K + (U – L + S) / S * N
```

- IVSub with Loop Normalization

```
I = 1
DO i = 1, (U-L+S)/S, 1
   K = K + N
   … = A (K)
   I = I + 1
ENDDO
```

⇒

```
I = 1
DO i = 1, (U – L + S) / S, 1
   … = A (K + i * N)
ENDDO
K = K + (U – L + S) / S * N
I = I + (U – L + S) / S
```

Summary

- Transformations to put more subscripts into standard form
  - Induction Variable Substitution
  - Loop Normalization
  - Related optimizations
    - Constant Propagation, redundancy elimination, deadcode elimination
- Do loop normalization before induction-variable substitution
  - Try eliminate symbolic loop steps
- Leave optimizations to compilers?