COMP 515: Advanced Compilation for Vector and Parallel Processors

Prof. Vivek Sarkar
Department of Computer Science
Rice University
vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP515

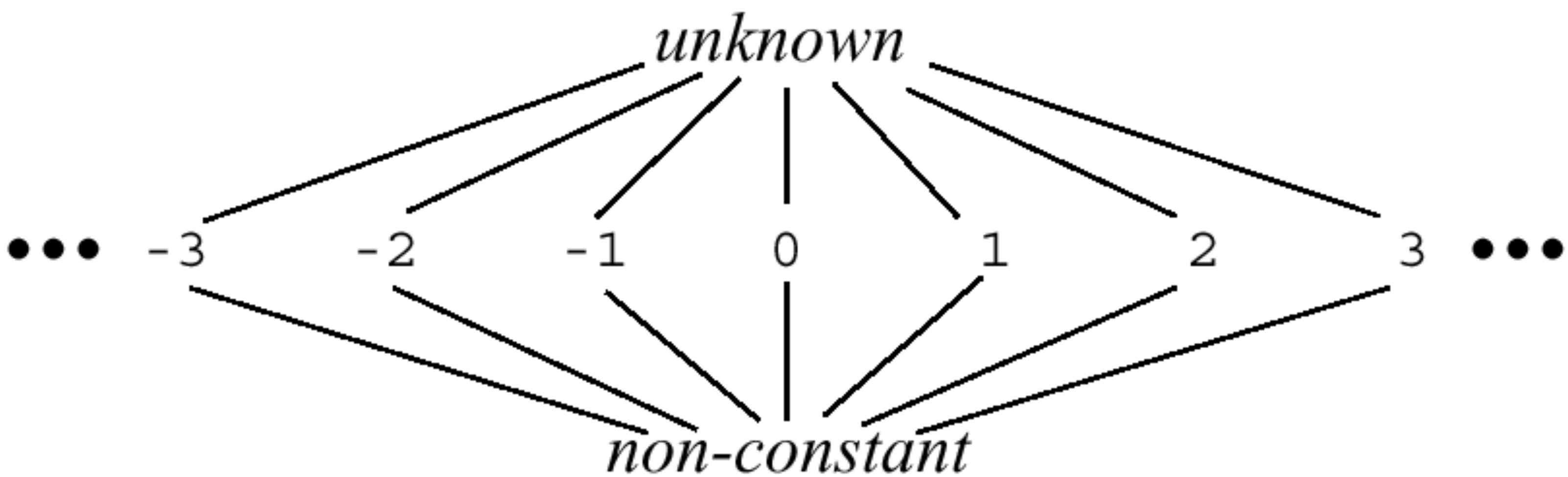
Homework #2 (Written Assignment)

- **Solve exercise 3.6 in book**
  - **This is case 4 of Lemma 3.3**
  - **Read Definitions 3.1, 3.2, 3.3 and Lemmas 3.1, 3.2, 3.3 before starting**
- **Due in class on Tuesday, Sep 22nd**

- Honor Code Policy: All submitted homeworks are expected to be the result of your individual effort. You are free to discuss course material and approaches to problems with your other classmates and the professors, but you should never misrepresent someone else's work as your own. If you use any material from external sources, you must provide proper attribution.

Constant Propagation

- **Replace all variables that have constant values at runtime with those constant values**
- **Constant propagation is a standard data flow analysis used by compilers that employs the lattice below**



Forward Expression Substitution

- **Example – the opposite of common subexpression elimination!**

```
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
```

Induction Variable Substitution

Definition: an auxiliary *induction* variable in a DO loop headed by DO I = LB, UB, S is any variable whose value can be correctly expressed as

$cexpr * I + iexpr_L$

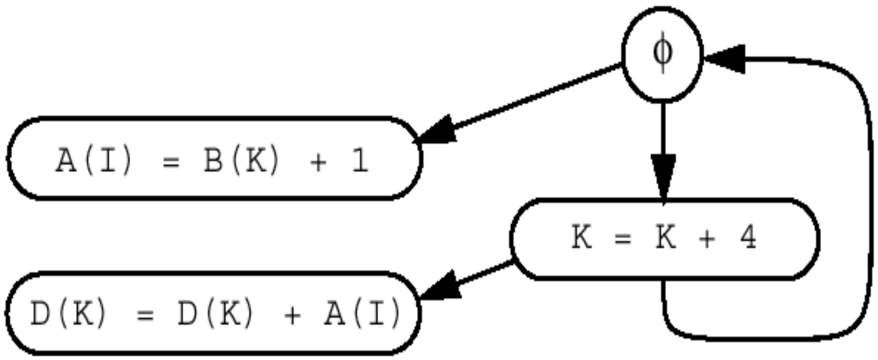
at every location L where it is used in the loop, where cexpr and $iexpr_L$ are expressions that do not vary in the loop, although different locations in the loop may require substitution of different values of $iexpr_L$

Induction Variable Substitution

Example:

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

Graphic from SSA-based induction variable substitution example:



becomes:

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

Induction Variable Substitution

- More complex example

```
DO I = 1, N, 2
    K = K + 1
    A(K) = A(K) + 1 ! K = I + init-value
    K = K + 1
    A(K) = A(K) + 1 ! K = I + 1 + init-value
ENDDO
```

- Alternative strategy is to recognize region invariance

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

IVSub without loop normalization

```
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
```

- Problems:
  - —Inefficient code
  - —Nonlinear subscript

IVSub with Loop Normalization

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

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

Summary

- **Transformations to put more subscripts into standard form**
  - Loop Normalization
  - Constant Propagation
  - Induction Variable Substitution
- **Do loop normalization before induction-variable substitution**
- **Leave optimizations to compilers**
  - Alternatively, perform preliminary transformations as look-aside analyses (then you're guaranteed to "do no harm")

Dependence: Theory and Practice

(Loop Distribution, Vectorization Algorithm)

Allen and Kennedy, Chapter 2

Loop Distribution

- **Can statements in loops which carry dependences be vectorized?**

```
      DO I = 1, N
S1        A(I+1) = B(I) + C
S2        D(I) = A(I) + E
      ENDDO
```

- **Yes! Dependence: $S_1 \, \delta_1 \, S_2$ can be converted to:**

```
     DO I = 1, N  ! A(2:N+1) = B(1:N) + C
S1        A(I+1) = B(I) + C
    END DO
    DO I = 1, N  ! D(1:N) = A(1:N) + E
S2        D(I) = A(I) + E
      ENDDO
```

Loop Distribution

```
    DO I = 1, N
S1      A(I+1) = B(I) + C
S2      D(I) = A(I) + E
    ENDDO
```

- **transformed to:**

```
DO I = 1, N
S1      A(I+1) = B(I) + C
ENDDO
DO I = 1, N
S2      D(I) = A(I) + E
ENDDO
```

- **leads to:**

```
S1      A(2:N+1) = B(1:N) + C
S2      D(1:N) = A(1:N) + E
```

Loop Distribution

- **Loop distribution fails if there is a cycle of dependences**

```
   DO I = 1, N
S1         A(I+1) = B(I) + C
S2         B(I+1) = A(I) + E
   ENDDO
```

$S_1 \ \delta_1 \ S_2$ **and** $S_2 \ \delta_1 \ S_1$

- **Another example:**

```
   DO I = 1, N
S1            B(I) = A(I) + E
S2            A(I+1) = C(I) + D
   ENDDO
```

Simple Vectorization Algorithm

```
procedure vectorize (L, D)
// L is the maximal loop nest containing the statement.
// D is the dependence graph for statements in L.
  find the set {S1, S2, ... , Sm} of maximal strongly-connected
  regions in the dependence graph D restricted to L (Tarjan);

  construct Lp from L by reducing each Si to a single node and
  compute Dp, the dependence graph naturally induced on Lp by D;

  let {p1, p2, ... , pm} be the m nodes of Lp numbered in an order
  consistent with Dp (use topological sort);

  for i = 1 to m do begin
    if pi contains a dependence cycle then
      generate a sequential DO-loop around the statements in pi;
    else
      directly rewrite pi in vector notation, vectorizing it with
      respect to every loop containing it;
  end
end vectorize
```

Problems With Simple Vectorization

```
      DO I = 1, N
               DO J = 1, M
S1                     A(I+1,J) = A(I,J) + B
               ENDDO
      ENDDO
```

- Dependence from S_1 to itself with d(i, j) = (1,0)
- Key observation: Since dependence is at level 1, we can vectorize the inner loop!
- Can be converted to:

```
      DO I = 1, N
S1       A(I+1,1:M) = A(I,1:M) + B
      ENDDO
```

- The simple algorithm does not capitalize on such opportunities

Worksheet (Lecture 7)

Name: ______________________ Netid: ______________________

```
      DO I = 1, N
S1          B(I) = A(I) + C
S2          A(I+1) = C(I) + D
   ENDDO
```

1. Compute the dependence graph for the above loop nest
2. Is it possible to distribute the loops around S1 and S2?
3. If your answer to #2 was yes, show the final code after loop distribution? (Don't worry about vectorization)

COMP 515 Projects

- Yuhan Peng, Maggie Tang
  - DFGL transformations and OpenCL generation
- Prasanth Chatarasi
  - Polyhedral extensions for data race detection
- Lucas Martinelli, Jonathan Sharman,
  - Exploration of dependences and transformations in higher level OO languages, with a focus on C++ language and libraries (RAJA, Kokkos)
- Jack Feser
  - Exploration of ILP solvers for dependence analysis
- Pete Curry, Lung Li
  - OpenCL transformations for Digital Signal Processors
- Zhipeng Wang
  - Memory Hierarchy Management for iterative graph structures.