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.

Dependence Testing

Allen and Kennedy, Chapter 3 (contd)

Banerjee Inequality (Recap)

- **Theorem 3.3 (Banerjee). Let D be a direction vector, and h be a dependence function. h = 0 can be solved in the region R iff:**

$$\sum_{i=1}^{n} H_i^-(D_i) \le b_0 - a_0 \le \sum_{i=1}^{n} H_i^+(D_i)$$

Proof: Immediate from Lemma 3.3 and the IMV.

Example of using Banerjee Inequality (Recap)

```
DO I = 1, N
   DO J = 1, M
      DO K = 1, 100
         A(I,K) = A(I+J,K) + B
      ENDDO
   ENDDO
ENDDO
```

Testing (I, I+J) for D = (=,<,*):

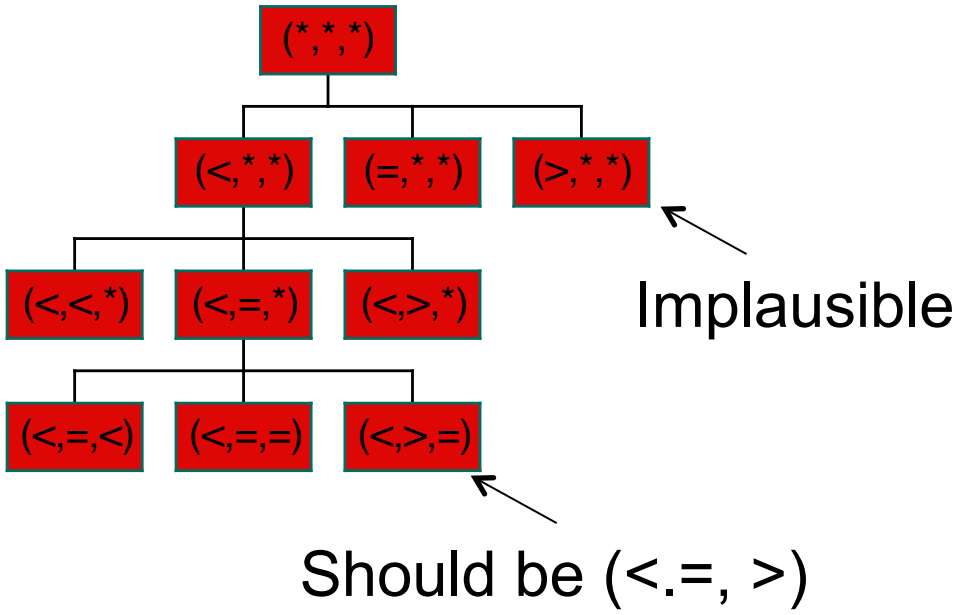
$$H_1^-(=) + H_2^-(<) = -(1-0)^- N + (1-1)^+ 1 - (0^- + 1)^+ (M-1) + [(0^- + 1)^- + 0^+]1 - 1 = -M \le 0$$

$$\le H_1^+(=) + H_2^+(<) = (1-1)^+ N - (1-1)^- 1 + (0^+ - 1)^+ (M-1) - [(0^+ - 1)^- + 0^-]1 - 1 \le -2$$

This is impossible, so the dependency doesn't exist.

Testing Direction Vectors

- Must test pair of statements for all direction vectors.
- Potentially exponential in loop nesting.
- Can save time by pruning:



Coupled Groups

- So far, we've assumed separable subscripts.
- We can glean information from separable subscripts, and use it to split coupled groups.
- Most subscripts tend to be SIV, so this works pretty well.

Delta Test

- Constraint vector *C* for a subscript group, contains one constraint for each index in group.
- The Delta test derives and propagates constraints from SIV subscripts.
- Constraints are also propagated from Restricted Double Index Variable (RDIV) subscripts, those of the form

$$< a_1 i + c_1, a_2 j + c_2 >$$

- See Figure 3.13 in textbook for Delta test algorithm
- Tiem for the worksheet!

Basic dependence algorithm (for a given direction vector)

```
Figure out what sort of subscripts we have

Partition subscripts into coupled groups

for each separable subscript

        test it using appropriate test

        if no dependence, we're done


for each coupled group

        use delta test

        if no dependence, we're done


return dependence otherwise
```

- For more advanced dependence tests, see the Omega Project http://www.cs.umd.edu/projects/omega/ and Polyhedral compiler frameworks

Preliminary Transformations

Chapter 4 of Allen and Kennedy

Overview

- Why do we need preliminary transformations?
- To create canonical representations of loop nests that simplify dependence testing
  - Requirements of dependence testing
    - Stride 1
    - Normalized loop
    - Linear subscripts
    - Subscripts composed of functions of loop induction variables
  - Higher dependence test accuracy
  - Easier implementation of dependence tests

An Example

- **Programmer optimized code**
  - **Confusing to smart 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
```

An Example

- Applying Induction-Variable Substitution (IVS)
  - Replace references to induction variables with functions of loop index for the purpose of dependence analysis

```
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
```

- In practice, induction variable information is often stored as "look-aside" information without actually transforming the code
  - Depends on whether optimizing back-end will strength-reduce the multiply operations

An Example

- **Second application of IVS**
  - **Remove all references to KI**

```
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
```

An Example

- Applying Constant Propagation
  - Substitute the constants

```
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
```

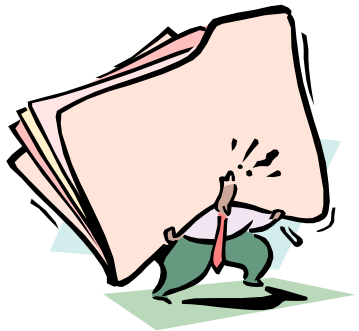
An Example

- Applying Dead Code Elimination
  - —Removes all unused code

```
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
```

Information Requirements

- Transformations need knowledge
  - Loop Stride
  - Loop-invariant quantities
  - Constant-values assignment
  - Usage of variables



Loop Normalization

- Transform loop so that
  - The new stride becomes +1 (more important)
  - The new lower bound becomes +1 (less important)
- To make dependence testing as simple as possible
- Serves as information gathering phase

- Examples for loops with non-unit strides
  - DO I = N, 1, -1 ➔
    - DO II = 1, N { I = N-II+1; … }
  - DO I = 2, 2*N, 2

Loop Normalization

- **Caveat**
  - 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 (<,>)

Loop Normalization

- **Caveat**
  - Consider interchanging loops
    - (<,=) becomes (=,<) OK
    - (<,>) becomes (>,<)

      Problem (as we will study later)

      Handled by another transformation (loop skewing)
  - What if the step size is symbolic?
    - Prohibits dependence testing
    - Workaround: use step size 1 (if we know step size is positive)

      Less precise, but allow dependence testing

Definition-use Graph

- Traditionally called Definition-use Chains
- Provides the map of variables usage
- Heavily used by preliminary transformations

Definition-use Graph

- Definition-use graph is a graph that contains an edge from each definition point in the program to every possible use of the variable at run time
- uses(b): the set of all variables used within the block b that have no prior definitions within the block
- defsout(b): the set of all definitions within block b that are not killed within the block
- killed(b): the set of all definitions that define variables killed by other definitions within block b

$$reaches(b) = \bigcup_{p \in P(b)} (defsout(p) \cup (reaches(p) \cap \neg killed(p)))$$

Dead Code Elimination

- Removes all dead code
- What is Dead Code ?
  - Code whose results are never used in any 'Useful statements'
- What are Useful statements ?
  - Are they simply output statements ?
  - Output statements, input statements, control flow statements, and their required statements
- Makes code cleaner
- Note that Dead Code is different from Unreachable Code
  - Unreachable code is code that can never be reached e.g., code for which all control conditions always evaluate to false

Dead Code Elimination

```
procedure eliminateDeadCode(P);
    // P is the procedure in which constants are to be propagated
    // Assume the availability of def-use chains for all the statements in P

    let worklist:= {absolutely useful statements};

    while worklist ≠ ∅ do begin
        x := an arbitrary element of worklist;
        mark x useful;
        worklist := worklist – {x};
        for all (y,x) ∈ defuse do
            if y is not marked useful then worklist := worklist ∪ {y};
    end
    delete every statement that is not marked useful;

end eliminateDeadCode
```

Worksheet (Lecture 6)

Name: _____ Netid: _____

```
DO I
        DO J
                DO K
                        A(J-I, I+1, J+K) = A(J-I,I,J+K)
                ENDDO
        ENDDO
ENDDO
```

- WORKSHEET: write the dependence equations for this example, and derive the distance vector
- First pass: establish $\Delta I = I' - I$ from second dimension
- Second pass: Propagate into first dimension to obtain ΔJ
- Third pass: Propagate into third dimension to obtain ΔK