CSc 553

Principles of Compilation

33 : Loop Dependence

Department of Computer Science
University of Arizona

collberg@gmail.com

Data Dependence Analysis

Data Dependence Analysis I

- Data dependence analysis determines what the constraints are on how a piece of code can be reorganized.
- If we can determine that no data dependencies exist between the different iterations of a loop we may be able to run the loop in parallel or transform it to make better use of the cache.
- For the code below, a compiler could determine that statement S_1 must execute before S_2, and S_3 before S_4. S_2 and S_3 can be executed in any order:

S_1: `A := 0;`
S_2: `B := A;`
S_3: `C := A + D;`
S_4: `D := 2;`

Dependence Graphs I

- There can be three kinds of dependencies between statements:

flow dependence

- Also, **true dependence** or **definition-use dependence**.

```
(i)    X   := ···
           .....
(j)    ··· := X
```

- Statement `(i)` generates (**defines**) a value which is used by statement `(j)`. We write `(i)` $\longrightarrow$ `(j)`.

anti-dependence

- Also, **use-definition dependence**.

```
(i)    ··· := X
           .....
(j)    X   := ···
```

Dependence Graphs II

- Statement (i) uses a value overwritten by statement (j). We write (i)$\not\longrightarrow$(j).

Output-dependence

- Also, **definition-definition dependence**.

```
(i)      X  := ···          .....
(j)      X  := ···
```

- Statements (i) and (j) both assign to (**define**) the same variable. We write (i)$\multimap\!\!\rightarrow$(j).
- Regardless of the type of dependence, if statement (j) depends on (i), then (i) has to be executed before (j).

Data Dependence Analysis I

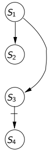
The Dependence Graph:

S_1: `A := 0;`

S_2: `B := A;`

S_3: `C := A + D;`

S_4: `D := 2;`



- In any program without loops, the dependence graph will be acyclic.
- Other common notations are

Flow	$\longrightarrow$	$\equiv$	δ	$\equiv$	δ^f
Anti	$\not\longrightarrow$	$\equiv$	$\bar{\delta}$	$\equiv$	δ^a
Output	$\multimap\!\!\rightarrow$	$\equiv$	δ°	$\equiv$	δ^o

Loop Fundamentals

Loop Fundamentals I

- We'll consider only perfect loop nests, where the only non-loop code is within the innermost loop:

```
FOR i_1 := 1 TO n_1 DO
  FOR i_2 := 1 TO n_2 DO
      ...
    FOR i_k := 1 TO n_k DO
      statements
    ENDFOR
      ...
  ENDFOR
ENDFOR
```

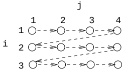
- The *iteration-space* of a loop nest is the set of *iteration vectors* (k-tuples): $\langle 1,1,1,\cdots\rangle,\cdots,\langle n_1,n_2,\cdots,n_k\rangle$.

Loop Fundamentals II

```
FOR i := 1 TO 3 DO
  FOR j := 1 TO 4 DO
    statement
  ENDFOR
ENDFOR
```

Iteration-space: $\{\langle 1,1\rangle, \langle 1,2\rangle, \langle 1,3\rangle, \langle 1,4\rangle,$
$\langle 2,1\rangle, \langle 2,2\rangle, \langle 2,3\rangle, \langle 2,4\rangle,$
$\langle 3,1\rangle, \langle 3,2\rangle, \langle 3,3\rangle, \langle 3,4\rangle\}.$

Represented graphically:



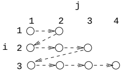
Loop Fundamentals III

- The iteration-space is often rectangular, but in this case it's *trapezoidal*:

```
FOR i := 1 TO 3 DO
  FOR j := 1 TO i+1 DO
    statement
  ENDFOR
ENDFOR
```

Iteration-space: $\{\langle 1,1\rangle, \langle 1,2\rangle,$
$\langle 2,1\rangle, \langle 2,2\rangle, \langle 2,3\rangle,$
$\langle 3,1\rangle, \langle 3,2\rangle, \langle 3,3\rangle, \langle 3,4\rangle\}$

Represented graphically:



Loop Fundamentals IV

- The index vectors can be lexicographically ordered. $\langle 1,1\rangle \prec \langle 1,2\rangle$ means that iteration $\langle 1,1\rangle$ precedes $\langle 1,2\rangle$.
- In the loop

```
FOR i := 1 TO 3 DO
  FOR j := 1 TO 4 DO
    statement
  ENDFOR
ENDFOR
```

  the following relations hold: $\langle 1,1\rangle \prec \langle 1,2\rangle$, $\langle 1,2\rangle \prec \langle 1,3\rangle$, $\langle 1,3\rangle \prec \langle 1,4\rangle$, $\langle 1,4\rangle \prec \langle 2,1\rangle$, $\langle 2,1\rangle \prec \langle 2,2\rangle$, $\cdots$, $\langle 3,3\rangle \prec \langle 3,4\rangle$.
- The iteration-space, then, is the lexicographic enumeration of the index vectors. Confused yet?

Loop Transformations

Loop Transformations I

- The reason that we want to determine loop dependencies is to make sure that loop transformations that we want to perform are legal.
- For example, (for whatever reason) we might want to run a loop backwards:

```
FOR i := 1 TO 4 DO              ⟹   FOR i := 4 TO 1 BY -1 DO
  A[i] := A[i+1] + 5                   A[i] := A[i+1] + 5
ENDFOR                               ENDFOR
```

- The original array is:

[1]	[2]	[3]	[4]	[5]
0	0	0	0	0

Loop Transformations II

- After the original loop the array holds:

[1]	[2]	[3]	[4]	[5]
5	5	5	5	0

- After the transformed loop the array holds:

[1]	[2]	[3]	[4]	[5]
20	15	10	5	0

- It is clear that, in this case, reversing the loop is not a legal transformation. The reason is that there is a data dependence between the loop iterations.
- In the original loop `A[i]` is read before it's assigned to, in the transformed loop `A[i]` is assigned to before it's read.

Loop Transformations III

- The dependencies are easy to spot if we unroll the loop:

```
S1:  A[1] := A[2] + 5
S2:  A[2] := A[3] + 5
S3:  A[3] := A[4] + 5
S4:  A[4] := A[5] + 5
       ⇑ Unroll
FOR i := 1 TO 4 DO
  A[i] := A[i+1] + 5
ENDFOR
       ⇓ Reverse & Unroll
S4:  A[4] := A[5] + 5
S3:  A[3] := A[4] + 5
S2:  A[2] := A[3] + 5
S1:  A[1] := A[2] + 5
```

- Graphically: $S_1 \to S_2 \to S_3 \to S_4$

Loop Dependencies I

- Hence, in this loop

```
FOR i := 1 TO 4 DO
  S1:  ··· := A[i+1]
  S2:  A[i] := ···
ENDFOR
```

there's an anti-dependence from S_1 to S_2: $S_1 \to S_2$

- In this loop

```
FOR i := 1 TO 4 DO
  S1:  A[i] := ···
  S2:  ··· := A[i-1]
ENDFOR
```

there's a flow-dependence from S_1 to S_2: $S_1 \to S_2$

Loop Dependence Analysis

Loop Dependence Analysis I

- Are there dependencies between the statements in a loop, that stop us from transforming it? A general, 1-dim loop:

```
FOR i := From TO To DO
  S1: A[f(i)] := ···
  S2: ··· := A[g(i)]
ENDFOR
```

- $f(i)$ and $g(i)$ are the expressions that index the array `A`. They're often of the form $c_1 * i + c_2$ (c_i are constants).
- There's a flow dependence $S_1 \longrightarrow S_2$, if, for some values of I^d and I^u, $\texttt{From} \le I^d, I^u \le \texttt{To}$, $I^d < I^u$, $f(I^d) = g(I^u)$, i.e. the two index expressions are the same.
- I^d is the index for the definition ($\texttt{A[}I^d\texttt{]:=}\cdots$) and I^u the index for the use ($\cdots\texttt{:=A[}I^u\texttt{]}$).

Loop Dependence Analysis II

Example

```
FOR i := 1 TO 10 DO
  S1: A[8*i+3] := ···
  S2: ··· := A[2*i+1]
ENDFOR
```

- $f(I^d) = 8 * I^d + 3$, $g(I^u) = 2 * I^u + 1$
- Does there exist $1 \le I^d \le 10$, $1 \le I^u \le 10$, $I^d < I^u$, such that $8 * I^d + 3 = 2 * I^u + 1$? If that's the case, then $S_1 \longrightarrow S_2$.
- Yes, $I^d = 1, I^u = 5 \Rightarrow 8 * I^d + 3 = 11 = 2 * I^u + 1$.
- There is a **loop carried** dependence between statement S_1 and S_2.

Simple Dependence Tests

The GCD Test

- Does there exist a dependence in this loop? I.e., do there exist integers I^d and I^u, such that $c * I^d + j = d * I^u + k$?

```
FOR I := 1 TO n DO
  S1: A[c*I+j] := ···
  S2: ··· := A[d*I+k]
ENDFOR
```

- $c * I^d + j = d * I^u + k$ only if $\gcd(c, d)$ evenly divides $k - j$, i.e. if $(k - j) \bmod \gcd(c, d) = 0$.
- This is a very simple and coarse test. For example, it doesn't check the conditions $1 \le I^d \le n$, $1 \le I^u \le n$, $I^d < I^u$.
- There are many other much more exact (and complicated!) tests.

The GCD Test – Example I

- Does there exist a dependence in this loop?

```
FOR I := 1 TO 10 DO
  S1: A[2*I] := ···
  S2: ··· := A[2*I+1]
ENDFOR
```

- $c * I^d + j = d * I^u + k$ only if $\gcd(c, d)$ evenly divides $k - j$, i.e. if $(k - j) \bmod \gcd(c, d) = 0$.
- $c = 2, j = 0, d = 2, k = 1$.
- $(1 - 0) \bmod \gcd(2, 2) = 1 \bmod 2 = 1$
- $\Rightarrow$ S_1 and S_2 are data independent! This should be obvious to us, since S_1 accesses even elements of A, and S_2 odd elements.

The GCD Test – Example II

```
FOR I := 1 TO 10 DO
  S1: A[19*I+3] := ···
  S2: ··· := A[2*I+21]
ENDFOR
```

- $c * I^d + j = d * I^u + k$ only if $\gcd(c, d)$ evenly divides $k - j$, i.e. if $(k - j) \bmod \gcd(c, d) = 0$.
- $c = 19, j = 3, d = 2, k = 21$.
- $(21 - 3) \bmod \gcd(19, 2) = 18 \bmod 1 = 0$
- $\Rightarrow$ There's a flow dependence: $S_1 \longrightarrow S_2$.
- The only values that satisfy the dependence are $I^d = 2$ and $I^u = 10$: $19 * 2 + 3 = 41 = 2 * 10 + 21$. If the loop had gone from 3 to 9, there would be no dependence! The gcd-test doesn't catch this.

The GCD Test – Example III

```
FOR I := 1 TO 10 DO
  S1: A[8*i+3] := ···
  S2: ··· := A[2*i+1]
ENDFOR
```

- $c * I^d + j = d * I^u + k$ only if $\gcd(c, d)$ evenly divides $k - j$, i.e. if $(k - j) \bmod \gcd(c, d) = 0$.
- $c = 8, j = 3, d = 2, k = 1$.
- $(1 - 3) \bmod \gcd(8, 2) = -2 \bmod 2 = 0$
- $\Rightarrow$ There's a flow dependence: $S_1 \longrightarrow S_2$.
- We knew this already, from the example in a previous slide. $I^d = 1, I^u = 5 \Rightarrow 8 * I^d + 3 = 11 = 2 * I^u + 1$.

Dependence Distance

Dependence Directions I

```
FOR I := 2 TO 10 DO
  S1: A[I] := B[I] + C[I];
  S2: D[I] := A[I] + 10;
ENDFOR
```

- On each iteration, S_1 will assign a value to A[i], and S_2 will use it.
- Therefore, there's a flow dependence from S_1 to S_2: $S_1\ \delta\ S_2$.
- We say that the **data-dependence direction** for this dependence is $\boxed{=}$, since the dependence stays within one iteration.
- We write: $S_1\ \delta_=\ S_2$.

Dependence Directions II

```
FOR I := 2 TO 10 DO
  S1: A[I] := B[I] + C[I];
  S2: D[I] := A[I-1] + 10;
ENDFOR
```

- On each iteration, S_1 will assign a value to A[i], and S_2 will use this value **in the next iteration**.
- E.g., in iteration 3, S_1 assigns a value to A[3]. This value is used by S_2 in iteration 4.
- Therefore, there's a flow dependence from S_1 to S_2: $S_1\ \delta\ S_2$.
- We say that the data-dependence direction for this dependence is $\boxed{<}$, since the dependence flows from i-1 to i.
- We write: $S_1\ \delta_<\ S_2$.

Dependence Directions III

```
FOR I := 2 TO 10 DO
  S1: A[I] := B[I] + C[I];
  S2: D[I] := A[I+1] + 10;
ENDFOR
```

- On each iteration, S_2 will use a value that will be overwritten by S_1 **in the next iteration**.
- E.g., in iteration 3, S_2 uses the value in A[4]. This value is overwritten by S_1 in iteration 4.
- Therefore, there's a anti dependence from S_2 to S_1: $S_2\ \overline{\delta}\ S_1$.
- We say that the data-dependence direction for this dependence is $\boxed{<}$, since the dependence flows from i to i+1.
- We write: $S_2\ \overline{\delta}_<\ S_1$.

Loop Nests

Loop Nests I

```
FOR I := 0 TO 9 DO
  FOR J := 1 TO 10 DO
    S1: ··· := A[I, J − 1]
    S2: A[I, J] := ···
  ENDFOR
ENDFOR
```

- With nested loops the data-dependence directions become **vectors**. There is one element per loop in the nest.
- In the loop above there is a flow dependence $S_2 \longrightarrow S_1$ since the element being assigned by S_2 in iteration I (A$[I, J]$) will be used by S_1 in the next iteration.
- This dependence is **carried** by the J loop.
- We write: $S_2\ \delta_{=,<}\ S_1$.

Loop Nests II – Example

```
FOR I := 1 TO N DO
  FOR J := 2 TO N DO
    S1: A[I, J] := A[I, J − 1] + B[I, J];
    S2: C[I, J] := A[I, J] + D[I + 1, J];
    S3: D[I, J] := 0.1;
  ENDFOR
ENDFOR
```

$\boxed{S_1\ \delta_{=,<}\ S_1}$ S_1 assigns a value to A$[I, J]$ in iteration (I, J) that will be used by S_1 in the next iteration $(I, J + 1)$. The dependence is carried by the J loop.

$\boxed{S_1\ \delta_{=,=}\ S_2}$ S_1 assigns a value to A$[I, J]$ in iteration (I, J) that will be used by S_2 in the same iteration.

$\boxed{S_2\ \overline{\delta}_{<,=}\ S_3}$ S_2 uses the value of D$[I + 1, J]$ in iteration (I, J). It will be overwritten by S_3 in the next I-iteration. The I-loop carries the dependence

Model

A Model of Dependencies

- Suppose we have the following loop-nest:

```
for i:=1 to x do
  for j := 1 to y do
    s1:  A[a*i+b*j+c,d*i+e*j+f] = ...
    s2: ...= A[g*i'+h*j'+k,l*i'+m*j'+n]
```

- Then there is a dependency between statements s_1 and s_2 if there exist iterations (i, j) and (i', j'), such that

$$\begin{aligned} a*i+b*j+c &= g*i'+h*j'+k \\ d*i+e*j+f &= l*i'+m*j'+n \end{aligned}$$

or

$$\begin{aligned} a*i-g*i'+b*j-h*j' &= k-c \\ d*i-l*i'+e*j-m*j' &= n-f \end{aligned}$$

A Model of Dependencies

- This is equivalent to an integer programming problem (a system of linear equations with all integer variables and constants) in four variables:

$$\begin{bmatrix} a & -g & b & -h \\ d & -l & e & -m \end{bmatrix} \times \begin{bmatrix} i \\ i' \\ j \\ j' \end{bmatrix} = \begin{bmatrix} k-c \\ n-f \end{bmatrix}$$

- If the loop bounds are known we get some additional constraints:

$$\begin{aligned} 1 \le i \le x, &\quad 1 \le i' \le x, \\ 1 \le j \le y, &\quad 1 \le j' \le y \end{aligned}$$

- In other words, to solve this dependency problem we look for integers i, i', j, j' such that the equation and constraints above are satisfied.

Homework

Exam I/a (415.730/96)

1. What is the gcd-test? What do we mean when we say that the gcd-test is *conservative*?
2. List the data dependencies ($\longrightarrow, \not\longrightarrow, \multimap$) for the loops below.

```
    FOR i := 1 TO 7 DO
S1:    ...           := A[2*i+1];
S2:    ...           := A[4*i];
S3:    A[8*i+3]      := ...;
    END;

    FOR i := 1 TO n DO
S1:    X             := A[2*i] + 5;
S2:    A[2*i+1]      := X + B[i+7];
S3:    A[i+5]        := C[10*i];
S4:    B[i+10]       := C[12*i] + 13;
    END;
```

Exam II (415.730/97)

- Consider the following loop:

```
FOR i := 1 TO n DO
    S1:   B[i] := C[i − 1] * 2;
    S2:   A[i] := A[i] + B[i − 1];
    S3:   D[i] := C[i] * 3;
    S4:   C[i] := B[i − 1] + 5;
ENDFOR
```

1. List the data dependencies for the loop. For each dependence indicate whether it is a **flow**- ($\longrightarrow$), **anti**- ($\not\longrightarrow$), or **output**-dependence ($\multimap$), and whether it is a **loop**-carried dependence or not.
2. Show the data dependence graph for the loop.

Summary

Readings and References

- Padua & Wolfe, *Advanced Compiler Optimizations for Supercomputers*, CACM, Dec 1996, Vol 29, No 12, pp. 1184–1187, http://www.acm.org/pubs/citations/journals/cacm/1986-29-12/p1184-padua/.

Summary I

- Dependence analysis is an important part of any parallelizing compiler. In general, it's a very difficult problem, but, fortunately, most programs have very simple index expressions that can be easily analyzed.
- Most compilers will try to do a good job on **common** loops, rather than a half-hearted job on all loops.
- Integer programming is NP-complete.

Summary II

- When faced with a loop

```
FOR i := From TO To DO
  S1: A[f(i)] := ···
  S2: ··· := A[g(i)]
ENDFOR
```

the compiler will try to determine if there are any index values I, J for which $f(I) = g(J)$. A number of cases can occur:

1. The compiler decides that $f(i)$ and $g(i)$ are too complicated to analyze. ⇒ Run the loop serially.
2. The compiler decides that $f(i)$ and $g(i)$ are very simple (e.g. `f(i)=i`, `f(i)=c*i`, `f(i)=i+c`, `f(i)=c*i+d`), and does the analysis using some built-in pattern matching rules. ⇒ Run the loop in parallel or serially, depending on the outcome.

Summary III

- contd.
  3. The compiler applies some advanced method to determine the dependence. ⇒ Run the loop in parallel or serially, depending on the outcome.
- Most compilers use pattern-matching techniques to look for important and common constructs, such as reductions (sums, products, min & max of vectors).
- The simplest analysis of all is a *name analysis*: If every identifier in the loop occurs only once, there are no dependencies, and the loop can be trivially parallelized:

```
FOR i := From TO To DO
  S1: A[f(i)] := B[g(i)]+C[h(i)];
  S2: D[j(i)] := E[k(i)]*F[m(i)];
ENDFOR
```