Dependence Testing

15-745 Optimizing Compilers
Spring 2006

Peter Lee

Loop parallelization

- In our previous lecture, we saw how locality can be improved for simple loops
- The transformations were based on knowledge of the **dependences** between loop iterations
- Dependence information is also critical for parallelizing loops
  - this is a fundamental goal for some applications and architectures

---

Parallelization example

```
for i = 1 to n
  for j = 2 to m
    b[i,j] = ...
    ... = b[i,j-1]
```

Iterations of the j loop must be executed sequentially.

But iterations of the i loop can be executed in parallel.

Determining this requires **dependence information**.

Types of dependences

- Reviewing from before...
- Four types of dependences
  - flow
  - anti
  - output
  - input

Flow dependence

```
1:  x = 1;
2:  y = x + 2;
3:  x = z - w;
 ...
4:  x = y / z;
```

Flow (aka **true**) **dependence**: Statement i precedes j, and i computes a value that j uses.

$1 \rightarrow^{t} 2$ and $2 \rightarrow^{t} 4$

Anti dependence

```
1:  x = 1;
2:  y = x + 2;
3:  x = z - w;
 ...
4:  x = y / z;
```

Anti dependence: Statement i precedes j, and i uses a value that j computes.

$2 \rightarrow^{a} 3$

Output dependence

```
1:  x = 1;
2:  y = x + 2;
3:  x = z - w;
 ...
4:  x = y / z;
```

Output dependence: Statement i precedes j, and i computes a value that j also computes.

$1 \rightarrow^{o} 3$ and $3 \rightarrow^{o} 4$

Input dependence

```
1:  x = 1;
2:  y = x + 2;
3:  x = z - w;
 ...
4:  x = y / z;
```

Input dependence: Statement i precedes j, and i uses a value that j also uses.

$3 \rightarrow^{i} 4$

Does not imply that i must execute before j

Dependences and renaming

- If $i \rightarrow^{?} j$, we say the dependence **flows** from i to j
  - i is the **source**, j is the **sink**
- The flow dependence, $i \rightarrow^{t} j$, is called the **true dependence**, because the other types are essentially programming style issues; they can be eliminated by renaming, e.g.:

```
1:   x = 1;
2:   y = x + 2;
3:   x1 = z - w;
  ...
4:   x2 = y / z;
```

Dependence graph

- Data dependences for a procedure are often represented by a data dependence graph
  - nodes are the statements
  - directed edges (labeled with t, a, o, or i) represented the dependence relations

---

Dependence testing

- Determining whether two statements are in a dependence relation is not easy
  - some readings will explore the issues for pointer-based structures
- But a lot of work has gone into understanding dependences for statements in loop bodies, particularly for array-based codes

---

A first example

```
1:   a[i] = b[i] + c[i];
2:   d[i] = a[i];
```

- There is a flow dependence, $1 \rightarrow^{t} 2$
- If we put this in a loop body, the dependence flows within the same iteration

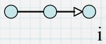
```
  for i = 2 to 4 {
1:   a[i] = b[i] + c[i];
2:   d[i] = a[i]; }
```

- We say that the dependence is **loop-independent**
  - aka: the **dependence distance** is 0
  - aka: the **dependence direction** is =

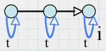
Iteration space

```
  for i = 2 to 4 {
1:   a[i] = b[i] + c[i];
2:   d[i] = a[i]; }
```

The iteration space for this loop: {2, 3, 4}



With dependences for a[] shown:

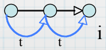


We write: $1 \rightarrow^{t}_{0} 2$ or $1 \rightarrow^{t}_{=} 2$

Example 2

```
  for i = 2 to 4 {
1:   a[i] = b[i] + c[i];
2:   d[i] = a[i-1]; }
```

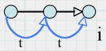
- There is a flow dependence, $1 \rightarrow^{t} 2$
- The dependence flows between instances of the statements in different iterations
  - this is a **loop-carried dependence**
- The dependence distance is 1
- The dependence direction is < (aka "positive")
- $1 \rightarrow^{t}_{1} 2$, or $1 \rightarrow^{t}_{<} 2$



Example 3

```
  for i = 2 to 4 {
1:   a[i] = b[i] + c[i];
2:   d[i] = a[i+1]; }
```

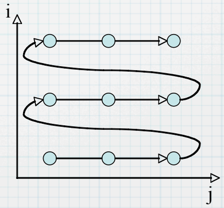
- There is an anti dependence, $2 \rightarrow^{a} 1$
- This is a loop-carried dependence
- The dependence distance is 1
- The dependence direction is < (aka "positive")
- $2 \rightarrow^{a}_{1} 1$, or $2 \rightarrow^{a}_{<} 1$



Example 4

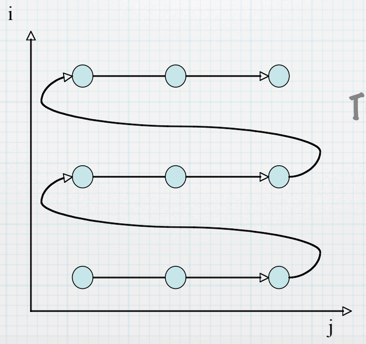
```
  for i = 2 to 4
    for j = 2 to 4
1:      a[i,j] = a[i-1,j+1];
```

- There is a flow dependence, $1 \rightarrow^{t} 1$
- This is a loop-carried dependence
- What is the dependence distance/direction?



Example 4

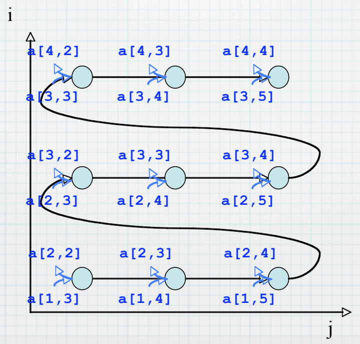
```
 for i = 2 to 4
   for j = 2 to 4
1:   a[i,j] = a[i-1,j+1];
```



The iteration space

Example 4

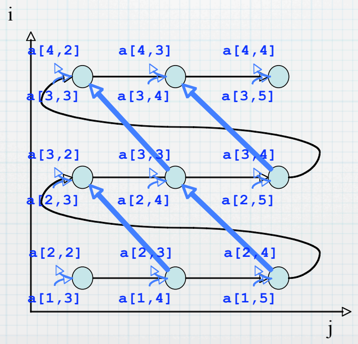
```
 for i = 2 to 4
   for j = 2 to 4
1:   a[i,j] = a[i-1,j+1];
```



The reads and writes of a[]

Example 4

```
 for i = 2 to 4
   for j = 2 to 4
1:   a[i,j] = a[i-1,j+1];
```



Dependence distance is (1,-1)

$1 \rightarrow t_{(1,-1)} 1$

$1 \rightarrow t_{(<,>)} 1$

The problem setup

For the time being, we restrict our attention to simple nested loops of the form:

```
for $i_1$ = $L_1$ to $U_1$
  for $i_2$ = $L_2$ to $U_2$
    ⋮
      for $i_n$ = $L_n$ to $U_n$
        a[$f_1(\vec{\imath})$, $f_2(\vec{\imath})$, ..., $f_d(\vec{\imath})$] = a[$g_1(\vec{\imath})$, $g_2(\vec{\imath})$, ..., $g_d(\vec{\imath})$];
```

with iteration vectors $\vec{\imath} = (i_1, i_2, ..., i_n)$

$$\vec{L} = (L_1, L_2, ..., L_n), \qquad \vec{U} = (U_1, U_2, ..., U_n), \qquad \vec{L} \leq \vec{U}$$

and f_i, g_i linear functions of the form

$$c_0 + c_1 i_1 + c_2 i_2 + \cdots + c_n i_n$$

The dependence test

- When does a dependence exist?
  - a dependence exists if:

there exist iteration vectors $\vec{k}$ and $\vec{\jmath}$ such that $\vec{L} \leq \vec{k} \leq \vec{\jmath} \leq \vec{U}$ and $f_i(\vec{k}) = g_i(\vec{\jmath})$, for $1 \leq i \leq d$.

Alternatively, $f_i(\vec{k}) - g_i(\vec{\jmath}) = 0$.

Dependence test example 1

```
 for i = 2 to 4 {
1:  a[i] = b[i] + c[i];
2:  d[i] = a[i-1]; }
```

- Are there iteration vectors i_1 and i_2, such that $2 \leq i_1 \leq i_2 \leq 4$ and $i_1 = i_2-1$?
- Yes: $i_1=2$, $i_2=3$ and $i_1=3$, $i_2=4$
- The distance vector is $i_2-i_1 = 1$
- The direction vector is sign(1) = <

Dependence test example 2

```
 for i = 2 to 4 {
1:  a[i] = b[i] + c[i];
2:  d[i] = a[i+1]; }
```

- Are there iteration vectors i_1 and i_2, such that $2 \leq i_1 \leq i_2 \leq 4$ and $i_1 = i_2+1$?
- Yes: $i_1=3$, $i_2=2$ and $i_1=4$, $i_2=3$
- The distance vector is $i_2-i_1 = -1$
- The direction vector is sign(-1) = >
- Is this possible?

Dependence test example 3

```
 for i = 1 to 10 {
1:  a[2*i] = b[i] + c[i];
2:  d[i] = a[2*i+1]; }
```

- Are there iteration vectors i_1 and i_2, such that $1 \leq i_1 \leq i_2 \leq 10$ and $2*i_1 = 2*i_2+1$?
- No! $2*i_1$ is even, whereas $2*i_2+1$ is odd
- So, there is no dependence

Dependence testing problem

- A classic problem in computer science
- Equivalent to an integer linear programming problem with 2n variables and n+d constraints
- An algorithm that finds two iteration vectors that satisfies these constraints is called a **dependence tester**
- This is an NP-complete problem, and so in practice the algorithms must be conservative

Dependence testers

- There are many dependence testers
- Each conservatively finds dependences
- Typically, a tester is designed to work only on specific kinds of indexing expressions
- Major testers include:
  - Lamport, GCD, Banerjee, I-test, power test, omega test, delta test, ...

Lamport test

- A simple test for index expressions involving a single index variable, and with the coefficients of the index variable all being the same
  - A[..., $b*i+c_1$, ...] = ...; ... = A[..., $b*i+c_2$, ...]
- Are there i_1 and i_2 such that $L \leq i_1 \leq i_2 \leq U$ and $b*i_1+c_1 = b*i_2+c_2$?

Lamport test, cont'd

- Are there i_1 and i_2 such that $L \leq i_1 \leq i_2 \leq U$ and $b*i_1+c_1 = b*i_2+c_2$?
- I.e., $i_2-i_1 = (c_1-c_2)/b$?
  - Note: integer solution exists only if $(c_1-c_2)/b$ is an integer
- Dependence distance is $d = (c_1-c_2)/b$, if $L \leq |d| \leq U$
  - $d>0$ means true dependence
  - $d=0$ means loop-independent dependence
  - $d<0$ means anti dependence

```
   for i = 1 to n
     for j = 1 to n
1:     a[i,j] = a[i-1,j+1];
```

i_1 = i_2-1?

b=1, c_1=0, c_2=-1

$(c_1-c_2)/b$ = 1

Dependence? Yes

Distance is 1

j_1 = j_2-1?

b=1, c_1=0, c_2=1

$(c_1-c_2)/b$ = -1

Dependence? Yes

Distance is -1

$1 \rightarrow^{t}_{(1,-1)} 1$

```
   for i = 1 to n
     for j = 1 to n
1:     a[i,2*j] = a[i-1,2*j+1];
```

i_1 = i_2-1?

b=1, c_1=0, c_2=-1

$(c_1-c_2)/b$ = 1

Dependence? Yes

Distance is 1

2*j_1 = 2*j_2-1?

b=2, c_1=0, c_2=1

$(c_1-c_2)/b$ = -1/2

Dependence? **No**

No dependence

GCD test

- Consider $\sum_{i=1}^{n} a_i x_i = c$
  - for a_i and c all integers
- An integer solution exists only if and only if gcd(a_1,a_2,...,a_n) divides c

---

```
   for i = 1 to n
1:   a[2*i] = b[i] + c[i];
2:   d[i] = a[2*i-1];
```

Are there i_1 and i_2 such that $1 \le i_1 \le i_2 \le 10$ and

2*i_1 = 2*i_2-1

or, equivalently

2*i_2 - 2*i_1 = 1?

There is an integer solution if and only if gcd(2,-2) divides 1

This is not the case, so no dependence

```
  for i = 1 to 10
1:   a[i] = b[i] + c[i];
2:   d[i] = a[i-100];
```

Are there i_1 and i_2 such that $1 \le i_1 \le i_2 \le 10$ and

$i_1 = i_2 - 100$

or, equivalently

$i_2 - i_1 = 100$?

There is an integer solution if and only if gcd(1,-1) divides 100

This is the case, so there is a dependence

But not really. GCD ignores loop bounds...

GCD test limitations

- Besides ignoring loop bounds, the GCD test also does not provide distance or direction information
- GCD is often 1, which ends up being very conservative

Dependence testing is hard

- Dependence testing is hard, both in theory and in practice
- Complications:
  - unknown loop bounds lead to false dependences

```
  for i = 1 to n
1:   a[i] = a[i+10];
```

Complications

- Aliasing
  - generally, must know that there is no aliasing in order for dependence testing to be conservative
- Triangular loops
  - generally requires addition of new constraints

```
  for i = 1 to n
    for j = 1 to i-1
1:     a[i,j] = a[j,i];
```

Complications, cont'd

- Many loops don't fit the mold exactly, but can be easily transformed to fit

```
   for i = 1 to n                for i = 1 to n
1:   x = a[i];        ──→     1:   x[i] = a[i];
2:   b[i] = x;                2:   b[i] = x[i];
```

```
   j = n-1
   for i = 1 to n                for i = 1 to n
1:   a[i] = a[j];     ──→     1:   a[i] = a[n-i];
2:   j = j-1;
```

Loop Parallelization

- A dependence is carried by a loop if that loop is the outermost loop whose removal eliminates the dependence

```
         for i = 2 to n-1
           for j = 2 to m-1
(=,=)        a[i,j] = ...;
             ... = a[i,j];

(=,<)        b[i,j] = ...;
             ... = b[i,j-1];

(<,=)        c[i,j] = ...;
             ... = c[i-1,j];
```

The outermost loop with a non-= direction carries the dependence

---

Parallelization

- The iterations of a loop may be executed in parallel if no dependences are carried by the loop

Example 1

(=,<)

```
for i = 2 to n-1
  for j = 2 to m-1
    a[i,j] = ...;
    ... = a[i,j-1];
```

The iterations of the j loop are sequential, but the i loop iterations are independent and can be executed in parallel

---

Example 2

(<,=)

```
for i = 2 to n-1
  for j = 2 to m-1
    a[i,j] = ...;
    ... = a[i-1,j];
```

The iterations of the j loop are parallel, but the i loop iterations must be executed sequentially

---

Example 3

(<,<)

```
for i = 2 to n-1
  for j = 2 to m-1
    a[i,j] = ...;
    ... = a[i-1,j-1];
```

The iterations of the j loop are parallel, but the i loop iterations must be executed sequentially

---

Loop interchange

- We saw earlier how loop interchange can improve the spatial locality of accesses

```
for j = 1 to n
  for i = 1 to n
    ...a[i,j] ...
```

```
for i = 1 to n
  for j = 1 to n
    ...a[i,j] ...
```

Loop interchange, cont'd

- Loop interchange can also increase the granularity of parallel computations

```
for i = 1 to n
  for j = 1 to n
    a[i,j] = b[i,j];
    c[i,j] = a[i-1,j];
```

(<,=)

```
for j = 1 to n
  for i = 1 to n
    a[i,j] = b[i,j];
    c[i,j] = a[i-1,j];
```

(=,<)

Recall: Interchange is legal when the interchanged dependences remain lexicographically positive