Dependence Analysis

Hwansoo Han
Dependence analysis

- Dependence
  - Control dependence
  - Data dependence

- Dependence graph
  - The way to represent dependences
  - Dependence types, latencies

- Usage
  - Instruction scheduling
  - Parallelization to exploit ...
    - Instruction level parallelism
    - Loop level parallelism
Dependence

- **Control dependence**
  - Relation defined on *control flow*
  - Different from execution order

- **Data dependence**
  - Relation defined on *data flow*
  - Flow / anti- / output / input dependence

- $S \delta T : T$ is dependent on $S$
  - Dependences constrain execution order
Control Dependence

- **S1 \( \delta^c \) S2**: S2 is control dependent on S1
  - if S1 determines whether S2 is executed

```
S1  br.eq  a, b  L1
S2  a ← b + c
S3  L1:  a ← a + e
```

- **S1 \( \delta^c \) S2**: ? ( )
- **S1 \( \delta^c \) S3**: ? ( )
- **S2 \( \delta^c \) S3**: ? ( )
Control Dependences around Branches & Joins

- Control dependences come from branch or join points in CFG

Diagram:

- Node S1 connected to S2 and L1
- Node L1 connected to S2 and S3
- Node S2 connected to S3

Equations:

- $S_1 \delta^c S_2 \ ? \ ( )$
- $S_1 \delta^c L_1 \ ? \ ( )$
- $L_1 \delta^c S_3 \ ? \ ( )$
- $S_2 \delta^c L_1 \ ? \ ( )$
- $S_2 \delta^c S_3 \ ? \ ( )$
- $S_1 \delta^c S_3 \ ? \ ( )$
Data Dependence

- $S_1 \delta^d S_2$: $S_2$ is data dependent on $S_1$
  - If $S_1$ and $S_2$ accesses the same variable AND
    $S_1$ precedes $S_2$ in program order

```
S1: a ← b + c
S1 $\delta^f$ S2 : a
S2: d ← a + e
S2 $\delta^a$ S3 : e
S3: e ← f + g
S2 $\delta^o$ S4 : d
S4: d ← g + h
S3 $\delta^i$ S4 : g
```
Types of Data Dependences

- **Flow dependence** ($\delta^f$)
  - true dependence
  - Can cause RAW hazard

- **Anti-dependence** ($\delta^a$)
  - register rename removes dependence
  - Can cause WAR hazard

- **Output dependence** ($\delta^o$)
  - register rename removes dependence
  - Can cause WAW hazard

- **Input dependence** ($\delta^i$)
  - does not constrain execution order
  - Can cause RAR (not hazard)
Example

Dependences on \( x \)

\[
\begin{align*}
S_1 & \quad \text{read } *, x, i, n \\
S_2 & \quad y = x + 10 \\
S_3 & \quad \text{if } (x \gt \text{gt.} 0) \text{ then} \\
S_4 & \quad z = y \times 2 \\
S_5 & \quad x = z - 1 \\
S_6 & \quad \text{else} \\
S_7 & \quad a[i] = y + 1 \\
S_8 & \quad i = i + 1 \\
S_9 & \quad \text{endif} \\
S_{10} & \quad a[1] = a[i-1] - 999 \\
S_{11} & \quad \text{do } j = 2, 10, 1 \\
S_{12} & \quad a[j] = a[j-1] \times y \\
S_{13} & \quad \text{enddo} \\
S_{14} & \quad \text{print } *, a[n], x
\end{align*}
\]
Dependence Graph

- DAG used to represent dependence relations.
- Allows the compiler to capitalize on graph algorithms to analyze dependences.
- Each edge may be annotated with
  - dependence types
  - execution latency
- Ex)

\[
\begin{align*}
S_1 & \delta S_2, S_1 \delta S_3, \\
S_1 & \delta^o S_5, S_1 \delta S_{14}, \\
S_2 & \delta^a S_5, S_3 \delta^a S_5, \\
S_5 & \delta S_{14}
\end{align*}
\]
Instruction Scheduling

- **Dependence** imposes execution order:
  - If $S_1 \delta S_2$, $S_1$ must execute earlier than $S_2$

- Scheduling constraint for control dependence:
  - cannot move instructions above/below branches
  - cannot move instructions above/below join points

```
b = 0;
if (x < 6) {
  y = 0;
  z = 5;
} else {
  y = 4;
  z = 5;
}
a = 1;
```

```javascript
b = 0;
if (x < 6) {
  y = 0;
  z = 5;
} else {
  y = 4;
  z = 5;
}
a = 1;
```

**Original code**

```javascript
b = 0;
if (x < 6) {
  y = 0;
  z = 5;
} else {
  y = 4;
  z = 5;
}
a = 1;
```

**OK**

```javascript
b = 0;
if (x < 6) {
  y = 0;
  z = 5;
} else {
  y = 4;
  z = 5;
}
a = 1;
```

**Error**

```javascript
b = 0;
if (x < 6) {
  y = 0;
  z = 5;
} else {
  y = 4;
  z = 5;
}
a = 1;
```

**Error**
Bending the Rule

- For some optimizations, control dependence may not be applied

- **Code motion / hoisting / sinking**
  - Moves instructions up/down the control flow as far as other dependences and resource constraints allow

- **Tail merging (Cross jumping)**
  - Moves instructions outside the basic blocks to their common successor if the instructions appear at the end of all these blocks
  - A special case of code sinking

```plaintext
original code

b = 0;
if (x < 6) {
  y = 0;
  z = 5;
} else {
  y = 4;
  z = 5;
}
a = 1;

OK

b = 0;
if (x < 6) {
  y = 0;
} else {
  y = 4;
  z = 5;
  a = 1;
}
```
VLIW Scheduling

- What does the dependence property imply to compiler techniques?
  - Some compiler techniques need to move instructions. (e.g., code motion)
  - Before they move instructions, they should identify dependences.

- Suppose that a compiler schedules this code on a 4-issue VLIW machine.

VLIW instructions

<table>
<thead>
<tr>
<th>$S_1$</th>
<th>$S_2$</th>
<th>$S_3$</th>
<th>$S_4$</th>
<th>$S_5$</th>
<th>$S_6$</th>
<th>$S_7$</th>
<th>$S_8$</th>
<th>$S_9$</th>
<th>$S_{10}$</th>
<th>$S_{11}$</th>
</tr>
</thead>
</table>

VLIW instructions (improved)

<table>
<thead>
<tr>
<th>$S_1$</th>
<th>$S_2$</th>
<th>$S_3$</th>
<th>$S_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_{58}$</td>
<td>$S_{10}$</td>
<td>$S_3$</td>
<td>$S_4$</td>
</tr>
<tr>
<td>$S_6$</td>
<td>$S_11$</td>
<td>$S_9$</td>
<td>$S_7$</td>
</tr>
</tbody>
</table>

Finding dependences helps the compiler to know where it may or may not move instructions.
Data Parallel Applications

- Scientific/multimedia applications
  - Focus on structured code (loops)
  - Array analysis (memory access pattern)

- Issues
  - Finding parallel loops
  - Transforming codes for better cache performance
Data Dependence Analysis

- Parallel loops?

```c
for (i=0; i<100; i++)
a[i] = (b[i-1] + b[i] + b[i+1]) / 3;
```

- Data dependence between loop iterations
- Data dependence analysis
Control dependence forces execution order among loop iterations (instances)
Dependences in loops (2/3)

- **Loop-independent** dependence
  - dependence within the same loop iteration

(loop code)

```
i = 1
L1:
a[i]=a[i-1]+1
b[i]=b[i+1]-1
...
i = i + 1
br.ne i,10,L1
```

loop independent anti-dependence
Dependences in loops (3/3)

- Loop-carried dependence
  - dependence across loop iterations
  - parallel loop: no loop-carried dependence

(loop code)

```
i = 1
L1:
a[i]=a[i-1]+1
b[i]=b[i+1]-1
...
i = i + 1
br.ne i,10,L1
```

(1\textsuperscript{st} iteration)

```
L1:
a[1]=a[0]+1
b[1]=b[2]-1
...
i = i + 1
br.ne i,10,L1
```

(2\textsuperscript{nd} iteration)

```
L1:
a[2]=a[1]+1
b[2]=b[3]-1
...
i = i + 1
br.ne i,10,L1
```
Iteration space

- Lexicographical order
- Distance vector: $d = i_2 - i_1$, if $i_1 \delta i_2$
- Direction vector

```plaintext
for i ← 0, N do
  for j ← 0, N do
    a[i, j] = a[i-1, j-1] + d;
    b[i, j] = b[i-1, j+1] + e;
    c[i, j] = c[i+1, j] + f
```

<table>
<thead>
<tr>
<th>distance</th>
<th>direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[] : &lt; 1, 1 &gt;</td>
<td>&lt; +, + &gt;</td>
</tr>
<tr>
<td>b[] : &lt; 1, -1 &gt;</td>
<td>&lt; +, - &gt;</td>
</tr>
<tr>
<td>c[] : &lt; 1, 0 &gt;</td>
<td>&lt; +, = &gt;</td>
</tr>
</tbody>
</table>
Data Dependence Test (1)

- Any two references access the same memory location?
- \( E i_1, i_2, \text{s.t. } 3i_1 - 5 = 2i_2 + 1, 1 \leq i_1, i_2 \leq 4 \) ?
- Finding integer solution for given integer coefficient inequality equations
  \( \rightarrow \) integer programming (NP-complete)
- Many tests exist
  - GCD test, Fourier-Motzkin elimination, Banerjee’s inequality

for \( i \leftarrow 1, 4 \) do
  \[ b[i] = a[3i-5] + 2; \]
  \[ a[2i+1] = 1 - i; \]

(solution)
\( i_1 = 4, \ i_2 = 3 \)
flow dependence
GCD Test

- GCD test
  - \[ a_0 + a_1 i_1 + \ldots + a_n i_n = b_0 + b_1 j_1 + \ldots + b_n j_n \]
  - \[ a_1 i_1 + \ldots + a_n i_n - b_1 j_1 - \ldots - b_n j_n = b_0 - a_0 \]
  - If GCD \((a_1, \ldots, a_n, b_1, \ldots, b_n)\) does not divides \(b_0 - a_0\), no integer solution exists

- Weakness
  - Does not consider index ranges
  - Only deals with affine functions of loop indexes

```plaintext
for i_1 \leftarrow 1, N_1
  for i_2 \leftarrow 1, N_2
    \ldots
      for i_n \leftarrow 1, N_n
        A[\ldots, a_0 + a_1 i_1 + \ldots a_n i_n, \ldots] = A[\ldots, b_0 + b_1 i_1 + \ldots b_n i_n, \ldots]
```
Fourier-Motzkin Elimination (1)

- Decide satisfiability of conjunction of linear constraints over reals
- Earliest method for solving linear inequalities
- Discovered in 1826 by Fourier, re-discovered in 1936 by Motzkin
- Basic idea
  - Pick one variable and eliminate it
  - Continue until all variables but one are eliminated

A system of conjoined linear inequalities $A\vec{I} \leq \vec{b}$

$m$ constraints

\[
\begin{pmatrix}
  a_{11} & a_{12} & \ldots & \ldots & a_{1n} \\
  a_{21} & a_{22} & \vdots & \vdots & \vdots \\
  \vdots & \ddots & \ddots & \ddots & \vdots \\
  \vdots & \ddots & \ddots & \ddots & \vdots \\
  a_{m1} & \ldots & \ldots & a_{mn} & \vdots \\
\end{pmatrix}
\begin{pmatrix}
  x_1 \\
  x_2 \\
  \vdots \\
  \vdots \\
  x_n \\
\end{pmatrix}
\leq
\begin{pmatrix}
  b_1 \\
  b_2 \\
  \vdots \\
  \vdots \\
  b_n \\
\end{pmatrix}
\]

$n$ variables
Fourier-Motzkin Elimination (2)

- Assume eliminating $x_1$

  Find $B_{lo} \leq x_1 \leq B_{up}$

- We eliminate $x_1$

  Add $B_{lo} \leq B_{up}$

Example:

(1) $x_1 - x_2 \leq 0$
(2) $x_1 - x_3 \leq 0$
(3) $-x_1 + x_2 + 2x_3 \leq 0$
(4) $-x_3 \leq -1$

<table>
<thead>
<tr>
<th>Category?</th>
<th>Lower bound</th>
<th>Upper bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(4)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(5) $2x_3 \leq 0$ (from 1 and 3)
(6) $x_2 + x_3 \leq 0$ (from 2 and 3)
Parallel Loop

- Loop $L_i$ is parallel, if all $D = \langle d_1, \ldots, d_n \rangle$,
  - $\langle d_1, \ldots, d_{i-1} \rangle$ is lexicographically forward (i.e. LC dep. in outer loops) or
  - $\langle d_1, \ldots, d_i \rangle = 0$ (i.e. no LC dependence in $L_i$)

```
for i ← 0, N do
    for j ← 0, N do
        a[i, j] = a[i, j-1] + c;
```

Distance vector: $<0, 1>$
- i (outer loop) : parallel
- j (inner loop) : not parallel

```
for i ← 0, N do
    for j ← 0, N do
        a[i, j] = a[i+2, j] + c;
```

Distance vector: $<2, 0>$
- i (outer loop) : not parallel
- j (inner loop) : parallel

```
for i ← 0, N do
    for j ← 0, N do
        a[i, j] = a[i-1, j+1] + c;
```

Distance vector: $<-1, -1>$
- i (outer loop) : not parallel
- j (inner loop) : parallel
Summary

- **Instruction scheduling/reordering** is a compiler technique that optimizes execution time or resource utilization by moving instructions based on the facts:
  - If no dependence exists among instructions, they can be executed in any order.
  - Instructions that are independent from each other can be executed in parallel.

- **Parallelization** also uses dependence information to achieve better performance by scheduling instructions for concurrent execution (e.g., VLIW compiler).