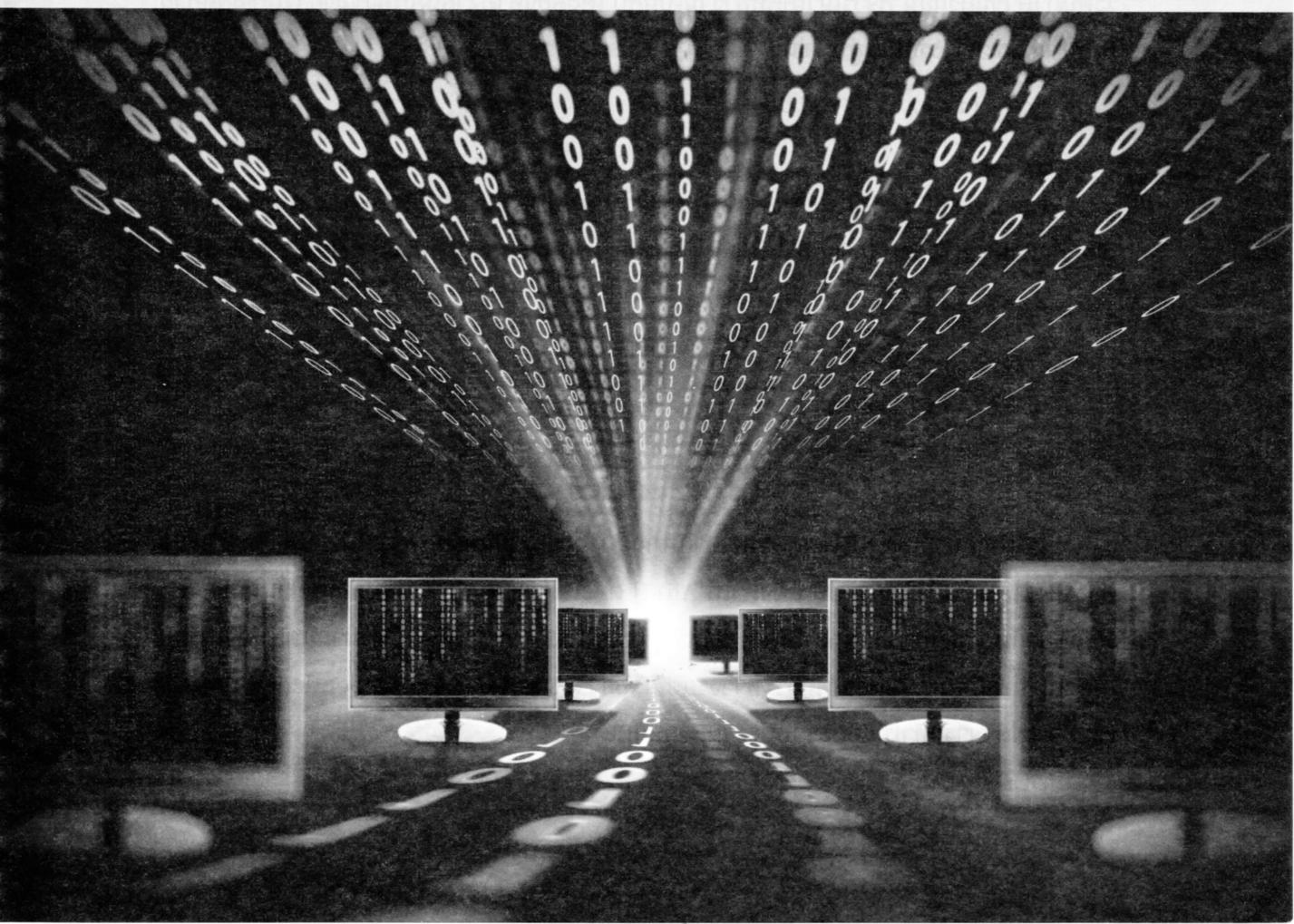


# 8

## เทคนิคการใช้ซอฟต์แวร์ร่วมกับ ฮาร์ดแวร์สนับสนุนการทำไปไลน์



ในการทำงานแบบบนหน้าจอ นอกจากจะต้องการฮาร์ดแวร์ที่รองรับการทำงานแบบบนหน้าจอแล้ว ยังต้องการการสนับสนุนจากซอฟต์แวร์ โดยเฉพาะในส่วนของตัวแปลงภาษา ซึ่งจะได้กล่าวถึงในบทนี้

ในบทนี้จะกล่าวถึงเทคนิคต่างๆ ที่เกี่ยวข้องห้งฮาร์ดแวร์และซอฟต์แวร์ที่ใช้ในการเพิ่มประสิทธิภาพการทำงานแบบบนหน้าจอ

## 8.1 การวิเคราะห์ความสัมพันธ์ของข้อมูล

ในการทำงานแบบบนหน้าจอ ได้แก่ ล่าถึงไปแล้วในบทก่อนหน้านี้ จะพบว่าความสัมพันธ์ของแต่ละคำสั่งขึ้นอยู่กับการใช้งานของข้อมูลในรีจิสเตอร์ ซึ่งถูกสร้างมาจากการคำสั่งในภาษาชั้นสูงอีกชั้นหนึ่ง

ในขั้นต้น เราจะต้องรู้จักการวิเคราะห์และมองหาความเป็นไปได้ในการทำงานแบบบนหน้าจอในระดับภาษาชั้นสูงก่อน พิจารณาโดยที่ทำงานเป็นลูป โดยมองหาความสัมพันธ์ของการใช้ข้อมูลภายในลูป และระหว่างรอบในลูป เช่น

```
for (i=1; i <= 100; i++)
    x[i] = x[i] + s;
```

ในตัวอย่างนี้มีการใช้  $x[i]$  ซึ่งจะเห็นว่า  $x[i]$  ในแต่รอบคำนวนได้โดยไม่ขึ้นแก่กัน นั่นคือ การคำนวนหาค่า  $x[1] \dots x[1000]$  ทำได้โดยไม่ต้องรอถ้าหากมีทรัพยากรหน่วยคำนวนที่สามารถทำได้

```
for (i=1; i <= 100; i++)
{
    A[i+1] = A[i] + C[i]; /* S1 */
    B[i+1] = B[i] + A[i+1]; /* S2 */
}
```

ในโค้ดนี้ ประযิค S1 ใช้ผลการคำนวนจาก S1 ในรอบก่อนหน้า เพราะในรอบที่  $i$  จะคำนวนค่า  $A[i+1]$  ซึ่งอ่านได้จากการรอบที่  $i$  ซึ่งทำให้  $A[i+1]$  ต้องใช้ค่าจาก  $A[i]$  ในรอบก่อนหน้านั้นเอง และ เช่นเดียวกัน สำหรับ  $B[i]$  และ  $B[i+1]$  สำหรับประยิค S2 การใช้  $A[i+1]$  คำนวนโดย S1 ในรอบก่อนหน้านี้ ลักษณะความสัมพันธ์ระหว่างรอบนี้เรียกว่า Loop-Carried Dependency ลักษณะ เช่นนี้จะไม่สามารถทำให้แต่ละรอบทำงานพร้อมกันได้

แต่ในบางกรณี การมี Loop-Carried Dependency แต่ละรอบก็ยังทำงานพร้อมกันได้ เช่น

```
for (i=1; i <=100; i++)
{
    A[i] = A[i] + B[i]; /* S1 */
    B[i+1] = C[i] + D[i]; /* S2 */
}
```

ในที่นี้ประโยค S1 จะขึ้นกับประโยค S2 ในรอบก่อนหน้าในตัวแปร B[i] แต่ประโยค S2 ไม่ขึ้นกับประโยค S1 ที่ติดกัน ดังนั้น จึงไม่มีความสัมพันธ์ภายในรอบเดียวกัน ทำให้มิเกิดความสัมพันธ์แบบวงกลม (Cyclic Dependency) ลักษณะแบบนี้จึงเรียกว่ามีลำดับบางส่วน (Partial Order) เท่านั้น กรณีนี้จะยังทำงานแบบขนานได้ ถ้ามีการจัดโคลด์ในลูปใหม่เป็น

```
A[1] = A[1] + B[1]
for (i=1; i <= 99; i++)
{
    B[i+1] = C[i] + D[i]; /* S1 */
    A[i+1] = A[i+1] + B[i+1]; /* S2 */
}
```

หลังจากจัดโคลด์ใหม่แล้ว จะทำให้มิเกิดความสัมพันธ์ระหว่างรอบ และแต่ละรอบจะทำงานขนานกันได้

โดยหลักการแล้ว เราต้องกำจัดความสัมพันธ์ซึ่งประกอบด้วยสิ่งที่ต้องทำ เช่น

- การมีลำดับโคลด์ที่ติด โดยให้ดูว่าลูปใดทำงานแบบขนานได้ และกำจัดความสัมพันธ์ที่เกิดจากชื่อ (Name Dependence) (ซึ่งยกในกรณีของตัวแปรแบบพอยน์เตอร์)
- ถ้าวิเคราะห์ได้ว่าไม่มี Loop-Carried Dependency จึงสามารถขยายลูปและสามารถมองหาการทำงานแบบขนานระหว่างรอบได้

ในตัวอย่างข้างล่าง ประโยค 2 การอ้างถึงตัวแปร a แต่ไม่จำเป็นต้องไปเอาที่หน่วยความจำ แต่ไปເອົາຄ່າຈາກຮີຈິສເຕອຣ໌ທີ່ເກັບຄ່າຂອງ a ໄດ້ໂດຍຕຽນຈາກຜລຍອງປະໂຍດທີ່ 1

```

for (i=1; i <= 100; i++)
{
    a[i] = b[i] + c[i]; // ประยุคที่ 1
    d[i] = a[i] + e[i]; // ประยุคที่ 2
}

```

ในการนี้ข้างล่างจะมี Loop-Carried Dependence ซึ่งเราต้องวิเคราะห์ลักษณะของการเกิดในรูปแบบเวียนบังเกิดของการอ้างอิงให้ได้

```

for (i=1; i <= 100; i++)
{
    y[i] = y[i-1] + y[i]; // ประยุคที่ 1
}

```

หรือในกรณีข้างล่างจะมีการอ้างถึงข้อมูลใน 5 รอบก่อนหน้านี้ (จาก  $y[i-5]$ ) ลักษณะนี้เรียกว่ามีระยะห่างระหว่างความสัมพันธ์ระหว่างรอบ (Dependency Distance) เท่ากับ 5 ในประยุคที่ 1

```

for (i=1; i <= 1000; i++)
{
    y[i] = y[i-5] + y[i]; // ประยุคที่ 1
}

```

โดยที่ไปแล้ว ตัวแปลภาษาต้องการทราบความสัมพันธ์ตามทฤษฎีนั้น Array Index จะเรียกว่า Affine ถ้าสามารถเขียนในรูป  $ai + b$  โดยที่  $a, b$  เป็นค่าคงที่ ถ้าเป็นอาร์เรย์หลายมิติ นั้นจะเป็นลักษณะ Affine ถ้าแต่ละมิติเป็นรูปแบบ Affine ด้วย การหาความสัมพันธ์ระหว่างรอบ เป็นการหาว่าในลูปที่ประกอบด้วย Affine Index ก็ต่อเมื่อ Index ทั้งสองเท่ากันที่รอบใดภายในขอบเขตของลูปที่ทำงานอยู่

สมมติให้  $P[ai + b] = F(Q(ci + d))$ ; เมื่อรัน  $i$  ระหว่าง  $m$  ถึง  $n$

- ความสัมพันธ์ของการใช้ข้อมูลในอาร์เรย์จะเกิดขึ้นเมื่อมี 2 รอบ  $j, k$  เมื่อ  $m \leq j, k \leq n$  เมื่อ  $aj + b = ck + d$  คือ  $P$  จะเก็บค่าที่อ่านจาก  $Q$

ปกติแล้ว วิธีการหาความสัมพันธ์แบบนี้จะใช้วิธีการ GCD

ถ้ามี Loop-Carried Dependency จะหมายถึง  $GCD(a, c)$  หาร  $(d - b)$  ได้ลงตัว

```
for (i=1; i <= 1000; i++)
{
    x[2*i+3] = x[2*i]+2; //1
}
```

ตัวอย่างนี้  $a = 2$ ,  $b = 3$ ,  $c = 2$ ,  $d = 0$  และ  $\text{GCD}(a, c) = 2$  และ  $d - b = -3$  และ  $2 \bmod 3 = -1$  ไม่ได้ จึงไม่มีความสัมพันธ์ของการใช้ข้อมูลเกิดขึ้น

```
for (i=1; i <= 1000; i++)
{
    y[i] = x[i] / c; // S1
    x[i] = x[i] + c; // S3
    z[i] = y[i] + c; // S3
    y[i] = c - y[i]; // S4
}
```

ในลูปมีความสัมพันธ์ระหว่างกัน ดังนี้

- จากประโยชน์ S1 ไปประโยชน์ S3, จากประโยชน์ S1 ไปประโยชน์ S4 จัดเป็น True Dependency เนื่องจากตัวแปร Y[i]
- จากประโยชน์ S1 ไปประโยชน์ S2 ( $x[i]$ ) เป็น Antidependence หรือ WAR
- จากประโยชน์ S3 ไปประโยชน์ S4 ( $y[i]$ ) เป็น Antidependence หรือ WAR
- จากประโยชน์ S1 ไปประโยชน์ S4 ( $y[i]$ ) เป็น Output Dependence หรือ WAW  
ถ้าแปลงโค้ดเป็น

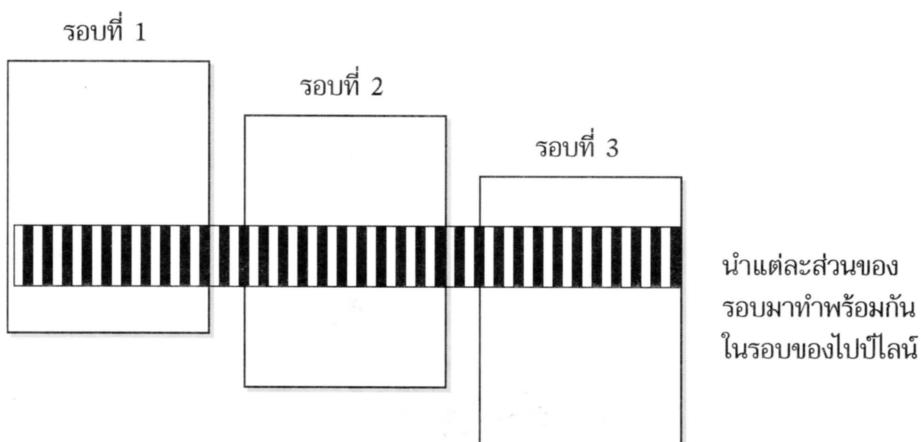
```
for (i=1; i <= 1000; i++)
{
    T[i] = x[i] / c; // เปลี่ยนชื่อ Y เป็น T กำจัด Output Dependence
    X1[i] = x[i] + c; // เปลี่ยนชื่อ x เป็น x1 กำจัด Anti Dependence
    z[i] = T[i] + c; // เปลี่ยนชื่อ Y เป็น T กำจัด Anti Dependence
    y[i] = c - T[i]; // S4
}
```

ตอนนี้ผลลัพธ์อยู่ที่ตัวแปร X1 ไม่ใช่อยู่ในตัวแปร X หลังจากทำส่วนนี้เสร็จ เราสามารถจะเปลี่ยนชื่อจากตัวแปร X1 เป็นตัวแปร X ได้โดยโดยไม่ต้องทำการคัดลอกค่าไป แต่ก็ยังมีกรณีอื่นๆ ที่เรายังไม่สามารถถวิเคราะห์ความสัมพันธ์ได้ เช่น

- การใช้พอยน์เตอร์เป็นอินเด็กซ์ในอาร์เรย์ เช่น A[\*ptr]
- เมื่อมีอินเด็กซ์ของอาร์เรย์เป็นอาร์เรย์ อีกชั้นหนึ่ง เช่น A[B[i]]
- เมื่อมีความสัมพันธ์เกิดขึ้นเฉพาะบางกรณีของข้อมูลเข้า เช่น กรณีที่อินเด็กซ์เป็นตัวแปรที่รับค่าเข้ามา
- เมื่อการออบตัวไม่ได้นั้นต้องการข้อมูลที่ขัดเจนว่าตัวแปรใดซึ่งกับตัวแปรใดบ้าง ไม่ใช่แค่มองว่ามีความเป็นไปได้ในการเกิดความสัมพันธ์เท่านั้น

## ■ 8.2 เทคนิคไปป์ไลน์ด้วยซอฟต์แวร์ (Software Pipelining)

เทคนิคไปป์ไลน์ด้วยซอฟต์แวร์เป็นเทคนิคที่คล้ายๆ กับการขยายลูปเพื่อมองหาคำสั่งที่มาทำงาน (Startup) ตอนต้นและโடดส่วนสุดท้ายหลังจากลูป (Finished Up) ซ้อนกันภายในลูปเดียวกัน โดยจะเอาคำสั่งจากลูปอื่นมาทำงานร่วมกัน ดังแสดงในรูปที่ 8.1 และจะต้องมีโடดส่วนที่เป็นโடดเริ่มต้น เป็นโടดปิดหัวท้ายลูปด้วย



รูปที่ 8.1 การนำคำสั่งในลูปต่างๆ มาทำงานพร้อมกันในไปป์ไลน์

### พิจารณาตัวอย่างโดดต่อไปนี้

```

Loop: LD      F0, 0 (R1)
      ADD.D  F4, F0, F2
      SD      F4, 0 (R1)
      ADDUI R1, R1, #-8
      BNE    R1, R2, Loop
  
```

โดยคำสั่ง ADD.D ย่อมาจากคำสั่งการบวกเลขเทคนิค และคำสั่ง ADDUI ย่อมาจากคำสั่งบวกจำนวนเต็มแบบ Unsigned ถ้าทำการขยายลูปออก 3 รอบเป็นดังนี้

รอบที่ 1	LD	F0, 0 (R1)
	ADD.D	F4, F0, F2
	<u>SD</u>	<u>F4, 0 (R1)</u>
รอบที่ 2	LD	F0, 0 (R1)
	<u>ADD.D</u>	<u>F4, F0, F2</u>
	SD	F4, 0 (R1)
รอบที่ 3	<u>LD</u>	<u>F0, 0 (R1)</u>
	ADD.D	F4, F0, F2
	SD	F4, 0 (R1)

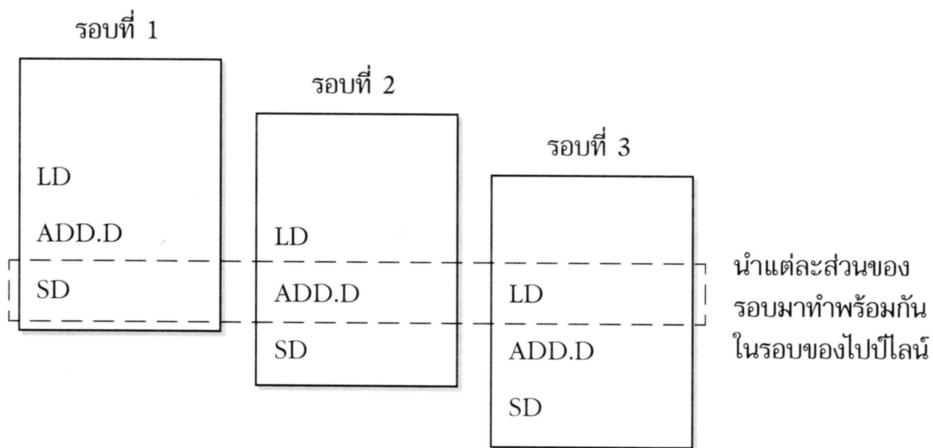
ในแต่ละรอบนั้น ทั้ง 3 คำสั่งต้องมีความสัมพันธ์กัน โดยต้องทำงานตามลำดับ แต่ถ้าพิจารณาคำสั่งที่ปิดเล่นให้นั้น เป็นการทำงานของแต่ละรอบทำกับข้อมูลที่ไม่ซึ้งกันแล้ว ก็จะทำให้ 3 คำสั่งมาประกอบเป็นลูปอันใหม่

```

Loop: SD      F4, 0 (R1)
      ADD.D  F4, F0, F2
      LD      F0, 0 (R1)
      ADDUI R1, R1, #-8
      BNE    R1, R2, Loop
  
```

คำสั่งแรก SD จะทำงานกับอาร์เรย์ M[i] คำสั่งต่อมา ADD.D จะทำงานกับอาร์เรย์ M[i-1] และคำสั่งต่อมา LD จะทำงานกับคำสั่ง M[i-2] โดยนี้ยังใช้ 3 คำสั่งแบบเดิม แต่เรียงลำดับต่างกัน เพราะทำงานกับอาร์เรย์คนละตำแหน่งกัน ดังนั้น จึงต้องเพิ่มโค้ดส่วนบนและส่วนล่างไปทึ่งจะอยู่นอกลูป โค้ดส่วนบนจะทำการ LD และ ADD.D ของตำแหน่ง M[1] ก่อนที่จะเข้ามาในลูปมา

ทำคำสั่ง SD M[1] และได้ดัดส่วนล่างจะทำการ ADD.D และ SD สำหรับอาร์เรย์ค่าสุดท้าย M[N]  
ให้เสร็จสิ้น การทำงานจะเป็นดังรูปที่ 8.2



ได้ใหม่นี้มีการทำงานดังข้างล่าง ผลจากคำสั่ง ADD จะถูกคำสั่ง SD ในรอบถัดไปใช้งาน  
ผลจากการทำคำสั่ง LD จะถูกนำไปบวกด้วยคำสั่ง ADD ในรอบต่อไปเพื่อกัน

รอบที่ 1	SD	F4, 0 (R1)
	ADD.D	F4, F0, F2
	LD	F0, 0 (R1)
	ADDUI	R1, R1, #-8
	SD	F4, 0 (R1)
รอบที่ 2	ADD.D	F4, F0, F2
	LD	F0, 0 (R1)
	ADDUI	R1, R1, #-8
	SD	F4, 0 (R1)
รอบที่ 3	ADD.D	F4, F0, F2
	LD	F0, 0 (R1)
	ADDUI	R1, R1, #-8

ข้างล่างนี้แสดงโค้ดหั้งหมดที่ได้รวมโค้ดนอกลูปและโค้ดท้ายลูปแล้ว

```

LD      F0,0 (R1)
ADD.D  F4,F0,F2
LD      F0,-8 (R1)
ADDUI  R1,R1,#-16
Loop: SD      F4,0 (R1)
       ADD.D  F4,F0,F2
       LD      F0,0 (R1)
       ADDUI  R1,R1,#-8
       BNE    R1,R2,Loop
       SD      F4,16 (R1)
       ADD.D  F4,F0,F2
       SD      F4,8 (R1)

```

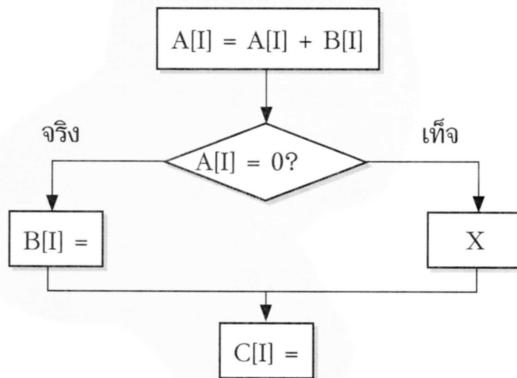
โค้ดจะเริ่มทำการ LD อาร์เรย์ 2 ค่าแรกก่อน M[1], M[2] และจะทำ ADD สำหรับ M[1] ก่อน เพื่อเข้าลูปจะทำการ SD สำหรับ M[1] ทำการบวกสำหรับ M[2] และทำการ LD สำหรับ M[3] แบบนี้ต่อไปเรื่อยๆ เมื่อบลูปจะทำการ SD ค่า M[N-1] และทำการบวกของอาร์เรย์ M[N] และจากนั้นทำการ SD อาร์เรย์ M[N] เพื่อให้เสร็จสิ้นหั้งอาร์เรย์

### ■ 8.3 เทคนิค Trace Scheduling

เป็นการเพิ่มเติมจากเทคนิคการขยายลูปเพื่อทำการทำงานแบบบนnanระหว่าง Conditional Branch เพื่อรับการ Issue ทีละหลายคำสั่งโดยทำการย้ายโค้ด หรือเรียกว่า Code Motion มี 2 ขั้นตอนดังนี้

1. Trace Selection เป็นการเลือก Trace ของคำสั่งเพื่อทำการหาลำดับการทำงานของบล็อกภายในซึ่งจะนำคำสั่งจากหลายๆ บล็อกเหล่านี้มารวมกันเป็น Trace
2. Trace Compaction เป็นการนำ Trace มารวมเป็นแพ็คของคำสั่งทำให้มีจำนวนแพ็คน้อยลง แต่ใช้จำนวนบิตในแต่ละแพ็คมาก

หั้งหมดนี้เรียกว่าเป็นการจัดลำดับโค้ดแบบภาพรวม (Global) เพื่อทำการหาลำดับของโค้ดที่สั้นและโดยรักษาความสัมพันธ์ของการใช้ข้อมูลและการควบคุมการทำงานไว้ตามโค้ดดังเดิม



รูปที่ 8.3 ตัวอย่าง Conditional Branch

สมมติว่าเลือก Trace ดังแสดงในรูปที่ 8.3

- จะทำการ Compact โดยการย้ายประโยชน์ให้ค่า โดยให้การให้ค่าตัวแปร  $B[i]$  และตัวแปร  $C[i]$  มาก่อนคำสั่งเงื่อนไขในการกระโดด
- การย้ายโค้ดในส่วนของ  $B[i] = \dots$  เป็นการคาดคะเน เพราะว่าประโยชน์นี้จะถูกทำเมื่อเงื่อนไขเป็นจริง และหากเงื่อนไขไม่เป็นจริง จะต้องระวังว่าการทำประโยชน์นี้ต้องไม่มีผลกับความถูกต้องของโปรแกรม เช่น การทำประโยชน์  $B[i]$  ต้องไม่เกิด Exception ต่างๆ ด้วย

ตัวอย่างในรูปที่ 8.3 มองเป็นโค้ดระดับล่างดังข้างล่าง

```

LD      R4, 0 (R1)      ; Load A
LD      R5, 0 (R2)      ; Load B
ADDUI  R4, R4, R5      ; Add
SD      R4, 0 (R1)      ; Store A
...
BNEZ   R4, Elsepart    ; ทดสอบเงื่อนไข A[i]
...
SD      0 (R2)          ; ประโยชน์ Then : B[i] ...
J      Join
Elsepart:
      X
  
```

Join:

SD            ... , 0 (R3)

; โคล์ดส่วน C[i]

พิจารณาโคล์ดและกรณีต่างๆ ดังนี้

### การย้ายโคล์ดส่วน B[i]

- ถ้าย้ายส่วน B[i] ไปไว้ก่อนหน้าคำสั่ง Branch ต้องไม่เกิด Exception ถ้าส่วนของ Else เป็นส่วนที่ได้ทำจริง
- การย้ายส่วน B[i] จะกระทบกับการให้ผลของข้อมูลภายในโปรแกรม ถ้าในโคล์ดส่วน X หรือ โคล์ดหลังจาก if มีการอ้างถึง B[i] นั้น

### ในการย้ายโคล์ดส่วน C[i]

- ถ้าย้ายโคล์ดส่วน C[i] ไปก่อนการ Join กันใน Trace จะทำให้โคล์ดส่วน C[i] มี Control Dependent กับคำสั่ง Branch ซึ่งจะทำให้มีไดรันเมื่อส่วน Else ต้องทำ จึงต้องย้าย โคล์ดส่วน C[i] ไปไว้ในส่วน Then และไว้เมื่อมีการ Join เพื่อมั่นเดิม
- เรามาจะย้ายโคล์ดส่วน C[i] ไปก่อนคำสั่ง Branch ได้ถ้าไม่ทำให้ความสัมพันธ์ของข้อมูล ผิดไป

การที่จะพิจารณาการย้ายการคำนวน B[i] มา จึงพิจารณาดังนี้

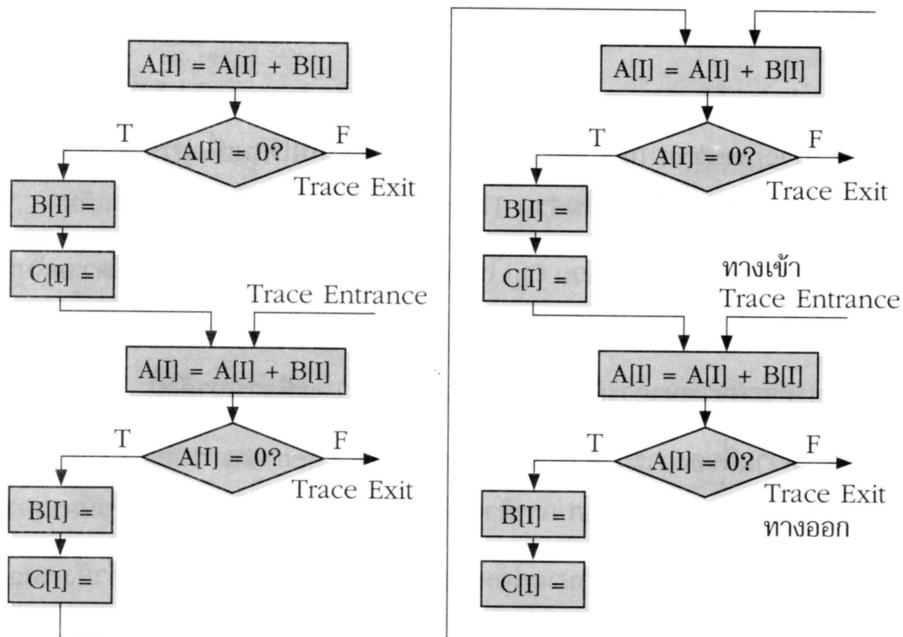
- ความต้องการการทำคำสั่งในแต่ละกรณีของ Then และกรณีของ Else ถ้าความต้องการ ทำส่วน Then มีมาก การย้ายโคล์ดส่วน B[i] มา ก่อนจะมีประโยชน์มากกว่า
- ค่าใช้จ่ายของการคำนวนโคล์ดส่วน B[i] ถ้ามีการย้ายมาไว้ก่อนคำสั่ง Branch
- การย้ายจะทำให้เวลาการทำงานสำหรับส่วนของ Then ลดลงหรือไม่ ถ้ากรณีส่วนของ Then เป็นส่วนที่รายการที่สุดในโคล์ด (Critical Path) ก็จะช่วยลดเวลาในการทำงานได้มาก
- ย้ายโคล์ดส่วนใน Then มาไว้แล้วจะมีประโยชน์สูงสุด
- ถ้ามีกรณีที่ต้องทำงานส่วนของ Else ขึ้น จำเป็นต้องมีโคล์ดเพิ่มเติม (เรียกว่า Compensation Code) เพื่อปรับแก้ผลให้ถูกต้องหรือไม่ และมีผลกับเวลาการทำงานรวมอย่างไร

```

while (...) {
    A[I] = A[I]+B[I];
    IF A[I] = 0
        B[I] = ...
    else
        X
    C[I] = ...
}

```

ถ้าทำการขยายลูปออกมา 4 รอบ จะได้การทำงานดังแสดงในรูปที่ 8.4



รูปที่ 8.4 การขยายลูป (Unroll) 4 รอบ

จะเห็นว่าเส้นทางการทำงานของคำสั่งที่จะนำจะทำให้ลื้นก็คือเส้นทางสำหรับ Then เพราะเป็นส่วนของ Critical Path ในการทำงาน

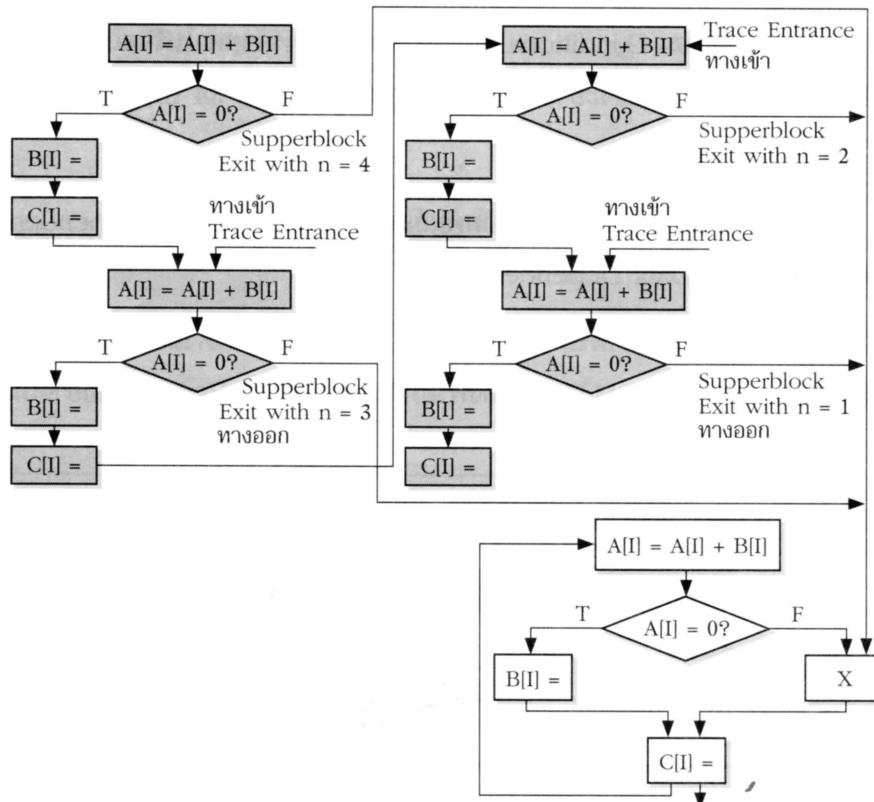
ปัญหาของการจัดลำดับของ Trace นี้คือการเข้า (Entry) และการออก (Exit) ระหว่าง Trace ต่างๆ ทำให้เกิดความยุ่งยาก ทำให้ตัวแปลภาษาต้องสร้างໂค้ดบางส่วนมาเติม

ถ้าพิจารณาเป็นแบบ Superblock โดยกำหนดให้ลักษณะของ Superblock คือ

- มีจุดเข้าจุดเดียว (Single Entry Point)
  - ประกอบด้วยบล็อกแบบพื้นฐาน (Extended Basic Block) หลายอัน
  - มีจุดออก (Exit Point) ของบล็อกหลายจุด

เพราะว่า Superblock มี Single Entry Point ทำให้การ Compact โค้ดง่ายกว่า ถ้าเป็นลูปที่เป็นลักษณะลูปแบบการนับ ก็จะทำให้เป็น Superblock แบบ Single Exit ด้วย

วิธีการสร้าง Superblock ทำได้โดย Tail Duplication โดยสร้างบล็อกของแต่ละ Trace หลักจาก Entry และใช้สูป Residual ในการจัดการรอบที่เหลือค้าง Superblock มีการ Exit ซึ่งจะเกิดจากการนิ่มคำสั่งที่ไม่ได้หมายไว้ถูกเลือกมาทำงาน (ในกรณีที่ทำงานผิด) นั่นเอง ดังแสดงในรูปที่ 8.5



รูปที่ 8.5 การทำงานเมื่อใช้แนวคิด Superblock

## ■ 8.4 คำสั่งแบบ Conditional Instruction และ Boosting

สำหรับเทคนิคต่างๆ ที่กล่าวมา เช่น การขยายลูป การทำไปเป็นของตัวเอง การจัดลำดับ Trace เป็นการกระทำในระหว่างการแปลโค้ด ถ้าพฤษฎิกรรมของคำสั่งกระโดดนั้นยังไม่ทราบล่วงหน้า จะต้องใช้เทคนิคอื่น เช่น การใช้ Conditional Instruction (หรือ Predicated Instruction) ช่วย ตัวแปลภาษาในการจำจัดคำสั่งแบบ Branch และขยายคำสั่งไปทำภายหลัง คำสั่ง Branch เป็นคำสั่งที่จะทำงานเมื่อเงื่อนไขเป็นจริง ถ้าเงื่อนไขนั้นเป็นเท็จ คำสั่งนั้นจะถูก跳过เป็น No-Op เช่นคำสั่ง

```
if (A == 0) { S=T; }
```

ถ้าให้รีจิสเตอร์ R4,R2,R3 เก็บค่า A,S,T ตามลำดับจะได้โค้ดดังนี้

BNEZ	R1, L
ADDU	R2, R3, R4

L1:

ถ้าใช้คำสั่งแบบ Conditional Move ซึ่งจะทำการย้ายค่าในรีจิสเตอร์ เมื่อโอเปอเรนด์ตัวที่ 3 เป็น 0 ดังรูปแบบ ซึ่งจะย้ายค่าใน R3 เข้าไปใน R2 เมื่อ R4 เป็น 0

```
CMOVZ R2, R3, R4
```

จะเท่ากับเป็นการเปลี่ยนความสัมพันธ์แบบเงื่อนไข (Control Dependence) เป็นความสัมพันธ์เชิงข้อมูล (Data Dependence)

พิจารณาตัวอย่างสำหรับ Superscalar ที่ Issue ได้ทั้งคำสั่งแบบเข้าถึงหน่วยความจำ 1 คำสั่ง และคำสั่งเกี่ยวกับ ALU 1 คำสั่งพร้อมกัน หรือทำการ Branch ได้อย่างเดียว

Memory Unit/Branch Unit		ALU Unit	
LW	R1, 40(R2)	ADD	R3, R4, R8
BEQZ	R10, L	ADD	R6, R3, R7
LW	R8, 20(R10)		
LW	R9, 0(R8)		

โดยดังกล่าวจะ Issue คำสั่ง LW และคำสั่ง ADD พร้อมกันในໄชเกิลแรก ส่งไปยังหน่วยจัดการหน่วยความจำและ ALU จากนั้นในໄชเกิลถัดไปจะ Issue คำสั่ง ADD อย่างเดียวไปทำงานใน ALU หน่วยจัดการหน่วยความจำ (Memory Unit) ก็จะว่างในໄชเกิลนี้

เนื่องจากหน่วยจัดการหน่วยความจำในໄชเกิลที่ 2 ยังว่างอยู่ ดังแสดงในช่องว่าง ถ้าใช้คำสั่งแบบ Conditional LW โดยจะอ่านค่าเมื่อโอเปอเรนด์ตัวที่ (R10) มีค่าเป็น 0 มาทำงานในเวลาที่เป็นการลดการ Stall ในไปป์ไลน์ที่เกิดขึ้นในเวลาที่ได้ ดังในคำสั่งที่ปิดเส้นใต้ LWC และคำสั่งนี้มีความสัมพันธ์กับคำสั่ง LW ตัวสุดท้ายในโอเปอเรนด์ R8 ที่ปิดเส้นใต้เข่นกัน

ดังนั้น เปรียบเสมือนการทำงานคำสั่ง LWC เป็นการคาดคะเน แต่ถ้ามีการทำนายผิดประสิทธิภาพจะเหมือนเดิม โดยคำสั่ง LW R9,0(R8) ยังจะโหลดค่าเดิมของ R8 มาใช้ แต่ยังต้องระวังเรื่องของการเกิด Exception ว่าถ้าคำสั่ง LW R9,0(R8) ทำให้เกิด Exception และ Exception นั้นควรจะเกิดหรือไม่ในความเป็นจริง เพราะคำสั่งนี้เป็นแบบคาดคะเนอยู่

Memory Unit/Branch Unit		ALU Unit
LW	R1, 40 (R2)	ADD R3, R4, R8
<u>LWC</u>	<u>R8, 20 (R10), R10</u>	ADD R6, R3, R7
BEQZ	R10, L	
LW	R8, 20 (R10)	
LW	R9, 0 (R8)	

ข้อจำกัดของ คำสั่งแบบ Conditional ได้แก่

1. คำสั่งแบบ Conditional จะถูกมองเป็น Nop ไปเมื่อไม่ได้ทำ และ Nop ยังกินเวลาทำงานของชีพียูอยู่
2. คำสั่งแบบ Conditional จะเป็นประโยชน์เมื่อการประเมินเงื่อนไขเกิดขึ้นเร็วและขั้นตอนการประเมินโดยการกระໂດแยกจากกัน
3. คำสั่งแบบ Conditional ใช้ไม่ได้กับกรณีเงื่อนไข Branch ขับช้อน เช่น เกิดจากการ And กันระหว่างเงื่อนไขอยู่ต่างๆ
4. คำสั่งแบบ Conditional ยังจะใช้จำนวนໄชเกิลมากกว่าคำสั่งที่ไม่เป็น Conditional อยู่

ในการอัมพลีเมนต์ทางชาร์ดแวร์ที่รองรับคำสั่งแบบนี้ ยังต้องใช้ชาร์ดแวร์และระบบปฏิบัติให้การทำงานร่วมกันเพื่อยกเลิกการเกิด Exception ในบางกรณี โดยอาจใช้บิตพิเศษที่เรียกว่า Poison Bit ติดไว้กับทุกรีจิสเตอร์ที่มีการถูกเขียนโดยคำสั่งแบบคาดคะเน (Speculated) เมื่อคำสั่นนั้นทำให้เกิด Exception และใช้วิธีการวนอกกว่าคำสั่นนั้นเป็นแบบคาดคะเน และใช้บัฟเฟอร์ทางชาร์ดแวร์เก็บผลลัพธ์จนกว่าคำสั่นนั้นจะเกิดขึ้นจริง จึงจะทำการเขียนผลลัพธ์ลงไป รวมไปถึงการจัดการ Exception ต้องระวังว่าการจัดการ Exception นั้นจะเป็นผลจริงได้ ถ้าคำสั่นนั้นเกิดขึ้นจริง ตัวอย่างเช่น

```
if (A==0) A=B; else A= A+4;
```

เมื่อ A อยู่ที่ 0(R4) และ B อยู่ที่ 0(R2) แปลงเป็นโค้ดได้เป็น

```
LD      R1,0 (R4)    ;load A
BNEZ   R1,L1        ;test
LD      R1,0 (R2)    ;if clause
J       L2           ;skip to else
L1:   DADDI  R1,R1,#4 ;else clause
L2:   SD     R1,0 (R4) ;store A
```

ถ้าใช้รีจิสเตอร์ R14 มาช่วยเพื่อหลีกเลี่ยงการเปลี่ยน R1 ก่อน ได้โค้ดดังข้างล่าง

```
LD      R1,0 (R4)    ;load A
LD      R14,0 (R2)   ;speculative load B
BEQZ   R1,L3        ;other branch of if
DADDI  R14,R1,#4   ;else clause
L3:   SD     R14,0 (R4) ;non speculative store
```

กรณีการใช้ Poison Bit เพื่อบอกว่าคำสั่งเป็น Speculative รวมทั้งการจัดการ Exception สำหรับตัวอย่างข้างล่าง

```
LD      R1,0 (R3)
LD*    R14,0 (R2)
BEQZ   R1,L3
DADDI  R14,R1,#4
L3:   SD     R14,0 (R3)
```

ถ้าคำสั่งหนึ่งๆ ใช้รีจิสเตอร์ที่บิต Poison ที่เขตเป็น 1 อยู่ การเขียนผลลัพธ์ใหม่ไปยังรีจิสเตอร์ที่เก็บค่านั้น ก็ทำให้รีจิสเตอร์นั้นมีบิต Poison ที่เป็น 1 ด้วย ดังนั้น ถ้าเกิด Exception ก็ยังเป็น Speculative ดังนั้น ถ้าคำสั่ง LD\* นี้เกิด Exception แบบ Termination ขึ้น บิต Poison ของ R14 จะเป็น 1 จนกว่าคำสั่ง LW ก็จะเกิดขึ้นจริง

วิธีการ Boosting เป็นการย้ายคำสั่งมาทำหลังจากคำสั่ง Branch ขึ้นมา และทำการเขต Flag ไว้ว่าเป็นคำสั่งแบบคาดคะเนโดยใช้อาร์ดแวร์ช่วยในการเปลี่ยนชื่อและทำการบันทึกเพื่อรองรับคำสั่งที่เป็นการ Boost ก็จะขึ้นอยู่กับผลของคำสั่ง Branch ที่อยู่คำสั่งต่อไป เมื่อถึงเวลาที่คำสั่ง Branch ทำงานและเมื่อมีการกระโดดเกิดขึ้นจริง ผลลัพธ์จะถูกเขียน (Commit) ไปที่รีจิสเตอร์ ดังตัวอย่างนี้จะไม่ต้องใช้รีจิสเตอร์เพิ่มแล้ว เพราะว่าผลลัพธ์ของคำสั่งแบบ Boost จะยังไม่เขียนไปยัง R1 จนกว่าคำสั่ง Branch จะถูกประเมินว่าต้องมีกระโดดจริง

LD	R1, 0 (R3)	; load A
LD+	R1, 0 (R2)	; boosted loaded B
BEQZ	R1, L3	; other branch of the if
DADDI	R1, R1, #4	; else
L3:	SD	R1, 0 (R3); nonspeculative store

## ■ 8.5 การทำ Multithreading

ในการใช้ ILP นั้นมีความยากในการตัดสินใจเลือก Benchmark มาใช้ในการวัดประสิทธิภาพของ ILP ความซับซ้อนทางอาร์ดแวร์ที่รองรับ และความซับซ้อนในตัวแплกภาษาของ ถ้าเพิ่มอาร์ดแวร์เพื่อต้องการเพิ่ม ILP ก็เท่ากับต้องเพิ่มค่าใช้จ่ายในส่วนของต้นทุน ดังนั้น ในการปรับปรุงที่เหมาะสมต้องพิจารณาความเหมาะสมของปริมาณอาร์ดแวร์ที่เพิ่มเติมไป และการปรับปรุงตัวแплกภาษาของเทียบกับ Speedup ที่จะได้เพิ่มขึ้น

ปัจจุบันเทคนิคทางอาร์ดแวร์ที่ใช้อยู่ ได้แก่

1. การเปลี่ยนชื่อรีจิสเตอร์เพื่อลด WAR, WAW กำหนดว่าจำนวนรีจิสเตอร์ไม่จำกัด
2. การทำนายการ Branch/Jump โดยมักสมมติว่าการทำนายถูกต้องและมีบันทึกเพื่อที่มีจำนวนที่พอเพียงในการเก็บคำสั่งที่เป็นแบบ Speculative

3. การวิเคราะห์การใช้หน่วยความจำตำแหน่งเดียวกัน โดยสมมติว่าค่าตำแหน่งในหน่วยความจำจะรู้ล่วงหน้าได้และสามารถถ่ายคำสั่ง Load ไปไว้ก่อนคำสั่ง Store ได้ถ้าตำแหน่งหน่วยความจำที่ใช้ค่อนลับตำแหน่งกัน

วิธีต่างๆ ที่ได้ก่อสร้างไปแล้ว ได้แก่

- การใช้เทคนิค Memory Disambiguation ใช้ชาร์ดแวร์เป็นตัวหาตำแหน่งของหน่วยความจำ
- การใช้ Speculation ในชาร์ดแวร์สำหรับการทำนายการ Branch ในขณะที่ทำงานแต่ละคำสั่งซึ่งตีกว่าการใช้ตัวแปลภาษา
- การใช้การจัดลำดับในซอฟต์แวร์เพื่อจัดลำดับคำสั่นทำงาน
- การวิเคราะห์หาความสัมพันธ์นั่นไม่ขึ้นกับชาร์ดแวร์ ซึ่งเป็นการปรับปรุงขั้นตอนการอوبติไมโครตัดและปรับปรุงโดยเพื่อให้ทำงานได้ดีขึ้น

ในการทำงานแบบบานานันนั้นยังมีอีกหลายมุมมอง เราอาจจะเห็นได้ชัดเจนในรูปแบบของงานใหญ่ๆ เช่น ระบบคลาวน์/ไฮบริด เน็ตเวิร์ก มีชาร์ดแวร์ให้บริการหลายๆ ตัวเพื่อบริการคลาวน์ การให้บริการหลายๆ บริการอาจจะทำได้พร้อมกัน ในที่นี้จะแบ่งได้ 2 แบบ คือ การทำงานแบบบานานะระดับของเทред (Thread-Level Parallelism /TLP) และการทำงานแบบบานานะระดับของข้อมูล (Data-Level Parallelism) สำหรับการทำงานแบบบานานะระดับเทредนั้น ตัวอย่างเช่น ในรูปแบบชาร์ดแวร์ข้างต้น แต่ละบริการที่ให้ก็จะเป็นเทред<sup>1</sup> ตัวหนึ่ง หรือถ้าเป็นแบบระดับข้อมูล ตัวอย่างเช่น การคำนวณการบวกระหว่าง 2 เวกเตอร์ โดยแต่ละค่าในแต่ละเวกเตอร์สามารถบวกกันได้แบบไม่มีข้อจำกัด รูปแบบนี้ได้ก่อสร้างไว้ในบทที่แล้ว

การจัดการทำงานแบบบานานะระดับข้อมูลจะดำเนินการตัวตัวเดียว กับข้อมูลจำนวนมาก ถ้าเปรียบเทียบระหว่าง TLP และ ILP นั้น ในการทำงานแบบ ILP นั้นจะมองหาคำสั่งที่จะทำงานแบบบานานกันได้ภายในโปรแกรมหนึ่งๆ โดยอาจจะมองหาในแต่ละส่วนของโปรแกรม เช่น ภายในลูป ภายในบล็อก แต่ในแบบ TLP จะใช้เทредทำงานหลายๆ คำสั่งพร้อมกัน โดยมีเป้าหมายเพื่อเพิ่ม Throughput และลดเวลาการประมวลผลโดยรวม

<sup>1</sup> ความหมายของเทредโดยพื้นฐาน คือ โปรแกรมที่มีคำสั่งและข้อมูลของตนเอง ซึ่งอาจจะทำงานส่วนหนึ่งของโปรแกรมหรืออาจจะเป็นทั้งโปรแกรมก็ได้ แต่ละเทредจะมีการเก็บสถานะได้แก่คำสั่งและข้อมูลที่ทำงาน ค่าในรีจิสเตอร์ PC และค่าในรีจิสเตอร์ของตนเอง

ในการใช้หลายๆ เทρดที่เรียกว่า Multi-threading นั้นจะใช้หลายๆ เทρดเพื่อให้หน่วยคำนวณร่วมกันภายในชิปปี้ โดยจะทำการหักห้อนการทำงานเข่นเดียวกันในแบบใบปีลайн แต่ละชิปปี้จะมีสถานะของแต่ละเทρดเก็บชุดของคำสั่งข้อมูล PC ค่าในรีจิสเตอร์ของแต่ละเทρดมีการใช้หน่วยความจำร่วมกัน ผ่านกระบวนการของหน่วยความจำสมீอันที่ใช้ฮาร์ดแวร์ช่วยเพื่อให้การสลับการทำงานของเทρดเร็วขึ้น (โดยประมาณใช้ 100 – 1,000 ไซเกิล) สำหรับการสลับเทρดนั้นจะมี 2 ระดับ ได้แก่ ระดับ Fine Grain และ Coarse Grain

ระดับ Fine Grain เป็นการสลับคำสั่งภายในแต่ละเทρด โดยจะทำการสลับເອກคำสั่งในเทρดมา ซึ่งทำให้การรับอาจจะประกอบด้วยหลายๆ เทρดของหลายๆ โปรแกรมพร้อมกันก็ได้ ทำให้เปิดโอกาสในการเลือกคำสั่งมาทำงานหักห้อนกันมากขึ้น โดยทั่วไปแล้ว การเลือกคำสั่งจะใช้รีวิ Round-Robin จะไม่พิจารณาเทρดที่กำลัง Stall อยู่ชิปปี้ ต้องสามารถสลับเทρดได้ในทุกรอบของสัญญาณนาฬิกา ในระดับ Fine Grain จะทำให้สามารถช่อน Stall ได้ทั้งแบบ Stall ยาวหลายไซเกิลและ Stall แบบสั้น แต่อาจจะทำให้การทำงานโดยรวมของแต่ละเทρดช้าลง เพราะว่าเทρdn อาจจะพร้อม แต่เวลาต้องถูกสลับออกไปตามการทำงานแบบ Round-Robin

ถ้ากรณีเทρดถูก Stall เนื่องจากการ Miss ของแคชใน L2 (ปัจจัยที่เกี่ยวกับแคชจะกล่าวถึงในบทต่อไป) ต้องทำการสลับเทρดทั้งเทρดออก วิธีนี้เรียกว่าเป็นแบบ Coarse Gain จะใช้สลับเทρดกรณีที่มี Stall นานๆ จะมีข้อดี คือ ลดเวลาในการสลับเทρดและไม่ต้องการวงจรที่ทำการสลับเทρดที่เร็ว เพราะอาจจะสลับเทρดไม่ปอย และจะไม่ทำให้เวลาการทำงานของแต่ละเทρดช้าลงในกรณีที่เทρdn นั้นพร้อมจะทำงานอยู่แล้ว แต่ข้อเสีย คือ ถ้ามี Stall แบบที่ใช้เวลาอย่างจะทำให้เสีย Throughput ไป เพราะต้องไปเริ่มเติมคำสั่งเข้าไปในใบปีลайнใหม่ ซึ่งใช้เวลา ซึ่งอาจทำให้ไม่คุ้มกรณีที่มี Stall แบบสั้นเกิดขึ้นบ่อย ถ้าใช้รีวิ Flush ใบปีลайнอาจคุ้มกว่า ดังนั้น วิธีนี้จะคุ้มกว่าเมื่อเวลาในการ Startup ใบปีลайн หรือเวลาในการทำให้ใบปีลайнเต็มอีกครั้งน้อยกว่าเวลาในการ Stall มากๆ

ในระบบปีจุบันจะใช้ทั้ง TLP และ ILP ร่วมกัน โดยใช้ TLP มาช่วยในการหาคำสั่งมาทำงานในใบปีลайн กรณีที่ Stall เกิดขึ้น เป็นการเพิ่มโอกาสของการทำงานแบบบานปลายโดยมองหาคำสั่งจากเทρดอื่นๆ มาทำงานแทน

ตัวอย่างข้างล่างเปรียบเทียบการทำงานแบบต่างๆ รูปที่ 8.6 เป็นกรณีมี 1 เทρดและมี 6 หน่วยคำนวณ ได้แก่ หน่วยจัดการหน่วยความจำ 2 หน่วย (M1, M2) หน่วย ALU 2 หน่วย (ALU1, ALU2) หน่วยทศนิยม 1 หน่วย (FP) และหน่วย Branch 1 หน่วย (BR) โดยในช่องที่ว่างจะหมายถึงหน่วยคำนวณนั้นว่าง

ไซเกิล	M1	M2	ALU1	ALU2	FP	BR
1						
2						
3						
4						
5						
6						
7						

รูปที่ 8.6 การใช้เทรดเดียวที่หน่วยคำนวนมี 6 หน่วย

ในรูปที่ 8.7 ถ้ามีการใช้ 2 เทรด ก็จะทำให้หน่วยคำนวนลดเวลาว่างลงไปได้

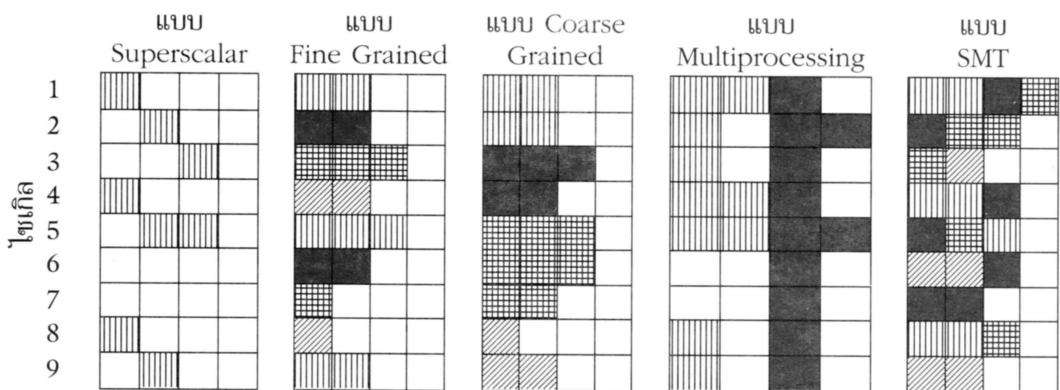
ไซเกิล	M1	M2	ALU1	ALU2	FP	BR
1						
2						
3						
4						
5						
6						
7						

เทรดที่ 1

เทรดที่ 2

รูปที่ 8.7 กรณีใช้ 2 เトレดสำหรับหน่วยคำนวน 6 หน่วย

ในการทำงานแบบ SMT จะให้ชีพียูทำการจัดลำดับคำสั่งในระหว่างເອົກສີຄົວຕ້ອງດ້ວຍຫາຽດແວຣ ໂດຍຈະໃຫ້ຮີຈິສເທອຣ໌ເສມ່ອນ (Virtual Register) ພລາຍໆ ຕັ້ງເພື່ອເກັບຄ່າຂໍ້ຄວາມອອນແຕ່ລະເທຣດ ມີ ປະເປດສີ່ຍັນຂໍ້ຮີຈິສເທອຣ໌ເພື່ອຂ່າຍຮະບູແລ່ລ່າມາຂອງແຕ່ລະຕັວແປຣທີ່ເໝາະສົມໃນ Data Path ແລະ ລົດ Dependency ໃນການປັບປຸງຂໍ້ອື່ນຂໍ້ອື່ນຈະມີຕາງໆເກັບການແປລັງຂໍ້ອື່ນແລ້ວແກັບ PC ຂອງແຕ່ລະເທຣດ ໃໃໝ່ Out-Of-Order Completion ເພື່ອທຳໄຫ້ເທຣດທຳການເສົ່າງໃນລັກຜະນະ Out-Of-Order ມີການໃຫ້ບັບເພົ່າງສໍາຮັບ Reorder Buffer ຂອງແຕ່ລະເທຣດໄດ້ເພື່ອເພີ່ມປະສິທິກາພກການໃຫ້ໜ່ວຍ ຄຳນວນຕ່າງໆ ແບບໜ້ານ



รูปที่ 8.8 ເປົ້າຍບໍ່ເຖິງການທຳການຂອງການຈັດລຳດັບໃນແຕ່ລະວູປແບບ

ໃນຮູປແບບ Superscalar ຈະໃຊ້ 1 ເທຣດທຳການເທົ່ານັ້ນ ແຕ່ແບບ Fine Gain ຈະໃຊ້ທີ່ 5 ເທຣດ ແລະ ສັບຄໍາສັ່ງການທຳການໃນແຕ່ລະຮອບ ໃນແບບ Coarse Gain ຈະໃຊ້ທີ່ 5 ເທຣດ ແຕ່ຈະສັບການທຳການເນື້ອແຕ່ລະເທຣດທຳໄດ້ໜ່າຍໆ ຄຳສັ່ງແລະເກີດ Stall ຊັ້ນ ໃນການແບບ Multiprocessing ຈະ ໄທແຕ່ລະເທຣດທຳການໃນແຕ່ລະບືໍ່ພືໍຍ ແລະ ໃນຮູປແບບ SMT ຈະເຫັນວ່າໃນໄເກີລ໌ນີ້ຈະທຳການໜ່າຍໆ ຄຳສັ່ງຈາກໜ່າຍເທຣດໄດ້ໂດຍໄມ້ຈຳກັດຍູ້ທີ່ຈຳນວນພືໍຍເມື່ອນກັບແບບ Multiprocessing

ເນື່ອງຈາກ SMT ຈະເປັນລັກຜະນະແບບ Fine Gain ດັ່ງນັ້ນ ຈະມີຜລກັບປະສິທິກາພບອອນແຕ່ລະເທຣດ ຄ້າເທຣດມີການ Stall ຈະມີການສັບເທຣດເກີດ ຈະເປັນການລົດ Throughput ໄປສໍາຮັບບາງເທຣດ ແລະ ຈະຕ້ອງໃຊ້ຈຳນວນຮີຈິສເທອຣ໌ມາກເພື່ອເກັບສົດການວອນແຕ່ລະເທຣດ ແຕ່ຈະໄມ້ຜລກັບຄວາມຍາວຮອບ ເວລາສັບຍຸາຜົນນາພິກາຂອງຮບຮົມ ໃນການ Issue ຄຳສັ່ງຈະມີທາງເລືອກໃນການ Issue ຄຳສັ່ງມາກເກີດ ເພວະຄຳສັ່ງອາຈະມາຈາກເທຣດອື່ນໄດ້ ໃນການ Commit ຈະມີການເລືອກຄຳສັ່ງທີ່ຈະ Commit ແລະ ຕ້ອງມີການຈັດການແຈ່ລະ TLB ໃຫ້ເໝາະສົມກັນເພື່ອໄມ້ໃຫ້ຜລຕ່ອປະສິທິກາພບແລະ ຄວາມຄຸກຕ້ອງ

ตัวอย่างของระบบแพร์เดียร์ ได้แก่ เครื่องสถาปัตยกรรมแบบ Power4 นั้นจะมีหน่วยการเอิกซ์คิวต์หลายตัว และทำให้เกิด Out-Of-Order Completion ได้ ส่วนในแบบ Power5 ซึ่งเป็นแบบ Multiple Issue โดยจะมี Data Path ที่รองรับ 2 แพร์เดียร์ ถ้าใช้ 4 แพร์เดียร์จะมีความต้องการในการใช้ทรัพยากร่วมกัน

ในตัว Power5 นั้นมีการปรับ Data Path และโครงสร้างเพื่อรองรับการทำงานแบบ SMT หลายอย่าง และขนาดของ Power5 Core จะใหญ่กว่า Power4 ประมาณ 24% เมื่อเพิ่มชาร์ดแวร์รองรับ SMT ตัวอย่างการปรับปรุง เช่น

- เพิ่ม Associativity ของแคชคำสั่งในระดับ L1 และบีฟเฟอร์ทำการแปลงแอดเดรสสำหรับคำสั่ง (Instruction Address Translation)
- เพิ่มคิวสำหรับคำสั่ง Load และ Store สำหรับแต่ละ Thread
- เพิ่มขนาดของแคช L2 (1.92 กับ 1.44 MB) และแคชระดับ L3
- เพิ่มหน่วย Instruction Prefetch และบีฟเฟอร์สำหรับแต่ละ Thread
- เพิ่มจำนวนรีจิสเตอร์สมேือนจาก 152 เป็น 240
- เพิ่มขนาดของคิวสำหรับการ Issue

ปัจจุบันการเพิ่มอัตราการ Issue คำสั่งเป็น 2 เท่า กล่าวคือ เป็น 3 – 6 คำสั่งต่อไซเกิล หรือ 6 – 12 คำสั่งต่อไซเกิลนั้นจะทำให้การเข้าถึงหน่วยความจำต้องเป็น 3 – 4 ครั้งต่อรอบ และทำให้จัดการคำสั่ง Branch ได้ 2 – 3 คำสั่งต่อรอบเวลา มีการเปลี่ยนชื่อรีจิสเตอร์และมีการเข้าถึงรีจิสเตอร์ถึง 20 ตัวต่อรอบเวลา และสำหรับการเฟตช์ ต้องทำการเฟตช์ถึง 12 – 24 คำสั่งต่อรอบ ความขับข้อนในการอิมเพลเม้นต์ชาร์ดแวร์สำหรับลีฟเฟิลนี้ทำให้ลดความถี่สัญญาณนาฬิกาลงไป เช่น ใน Itanium 2 จะมีความถี่ที่ชาแมวว่าต้องใช้กำลังมากก็ตาม กล่าวโดยรวมแล้ว เทคนิคต่างๆ ที่ทำให้เพิ่มประสิทธิภาพเป็นการเพิ่มการใช้กำลัง (Power) หรือพลังงาน (Energy) ทั้งสิ้น การใช้ชีพีย์แบบ Multiple-issue ก็ล้วนกินพลังงานทั้งสิ้น การ Issue คำสั่งหลายๆ คำสั่งพร้อมกันทำให้ชาร์ดแวร์เกตต่างๆ มีความขับข้อน มีจำนวนครั้งการ Switch ของทรานซิสเตอร์ในอัตราสูง เทียบกับอัตราการ Issue ที่สูงขึ้น และความแตกต่างนี้จะเป็นของว่างที่จะเพิ่มขึ้นต่อไปในอนาคต ตัวอย่างจะเห็นได้จากใน Itanium จะเห็นว่าแม้จะเพิ่ม ILP และจะเพิ่มความซับซ้อนของชาร์ดแวร์ และผังต้องการกำลังเพิ่มขึ้นอีกด้วย

ดังนั้น แผนที่จะเพิ่ม ILP จึงมีการหันมาใช้ TLP มากขึ้นในมัลติโพรเซสเซอร์แบบชิปเดียว (Single-Chip Multiprocessor) ดังจะเห็นได้จาก Power4 ของ IBM ซึ่งมีชิปยู Power3 จำนวน 2 โพรเซสเซอร์ และมีแคช L2 สำหรับ SUN, AMD และ Intel ก็หันมาสนใจมัลติโพรเซสเซอร์แบบชิปเดียวดังจะได้เห็นในเทคโนโลยีของ Multicore ในปัจจุบัน ขอบเขตการใช้ ILP และ TLP ยังไม่ชัดเจนนัก ขึ้นกับเป้าหมายการใช้งาน เช่น เป็น Desktop หรือเซิร์ฟเวอร์ เช่น ถ้าเป็นเซิร์ฟเวอร์ จะมีแนวโน้มการใช้แบบ Multi-Thread และ Desktop ใช้แบบเทรดเดียว เป็นต้น

## 8.6 สรุป

ในการทำงานแบบขนาดต้องการการการทำงานร่วมกันทั้งฮาร์ดแวร์และซอฟต์แวร์ ในซอฟต์แวร์ มีเทคนิคต่างๆ ที่ช่วยสนับสนุนการทำงานแบบขนาดระดับคำสั่ง และการเพิ่มรูปแบบชุดคำสั่งก็อาจ จะทำให้เปลี่ยนแปลงจำนวน Stall ที่เกิดขึ้น แต่ก็ต้องการการสนับสนุนของฮาร์ดแวร์ด้วยเช่นกัน

บทนี้ก่อสร้างเทคนิคขั้นสูงในการเพิ่มการทำงานแบบขนาดทั้งด้านฮาร์ดแวร์และซอฟต์แวร์ ตั้งแต่การประเมินความล้มเหลว การใช้เทคนิคไปป์ไลน์แบบซอฟต์แวร์ การใช้การจัดลำดับ Trace การใช้คำสั่งรูปแบบ Condition และการสนับสนุนทางฮาร์ดแวร์ การจัดการ Exception รวมไปถึงการใช้ Multi-Threading หลายรูปแบบในสถาปัตยกรรมปัจจุบัน

## คำダメท้ายบท

1. การทำงานไปปีลайнแบบของฟ์แวร์คืออะไร

2. Affine Index คืออะไร

3. จากโค้ดต่อไปนี้ จงหาความสัมพันธ์ของแต่ละคำสั่ง

```
for (i=0; I < 100; i++) {
    A[i] = B[i] + A[i-1];
    B[i] = B[i-2] + C[i];
    C[i] = A[i] + B[i];
}
```

4. จากโค้ดต่อไปนี้ จงขยายถูปอกรมา 2 รอบ และใช้ Trace Scheduling มาช่วยจัดลำดับ

```
while (1) {
    if (a[i] < b[i] )
        a[i] = b[i] + 1;
    i++;
    a[i] = c[i] + b[i];
}
```

5. จากโค้ดต่อไปนี้

```
for (i=0; I < 100; i++)
    A[i] = A[i] + B[i];
```

5.1 จงเขียนในรูปแบบของ MIPS

5.2 จงปรับโค้ดโดยใช้ไปปีลайнแบบของฟ์แวร์ และอธิบายโค้ดที่ปรับแล้ว

6. จงยกตัวอย่างการเพิ่มการทำงานแบบขนานด้วยกลไกทางชาร์ดแวร์และกลไกทางซอฟต์แวร์  
มาอย่างละ 2 แบบ

7. SMT ต่างกัน Superscalar อย่างไร
8. จงบอกข้อแตกต่างของ Fine Grain และ Coarse Grain TLP และมีข้อดี-ข้อเสียอย่างไร
9. Trace Scheduling คืออะไร การมองหา Trace ทำได้อย่างไร
10. Boosting ทำงานอย่างไร
11. คำสั่งแบบ Conditional จะทำให้การเปลี่ยนแปลง Data Path เป็นอย่างไร จงอธิบาย

