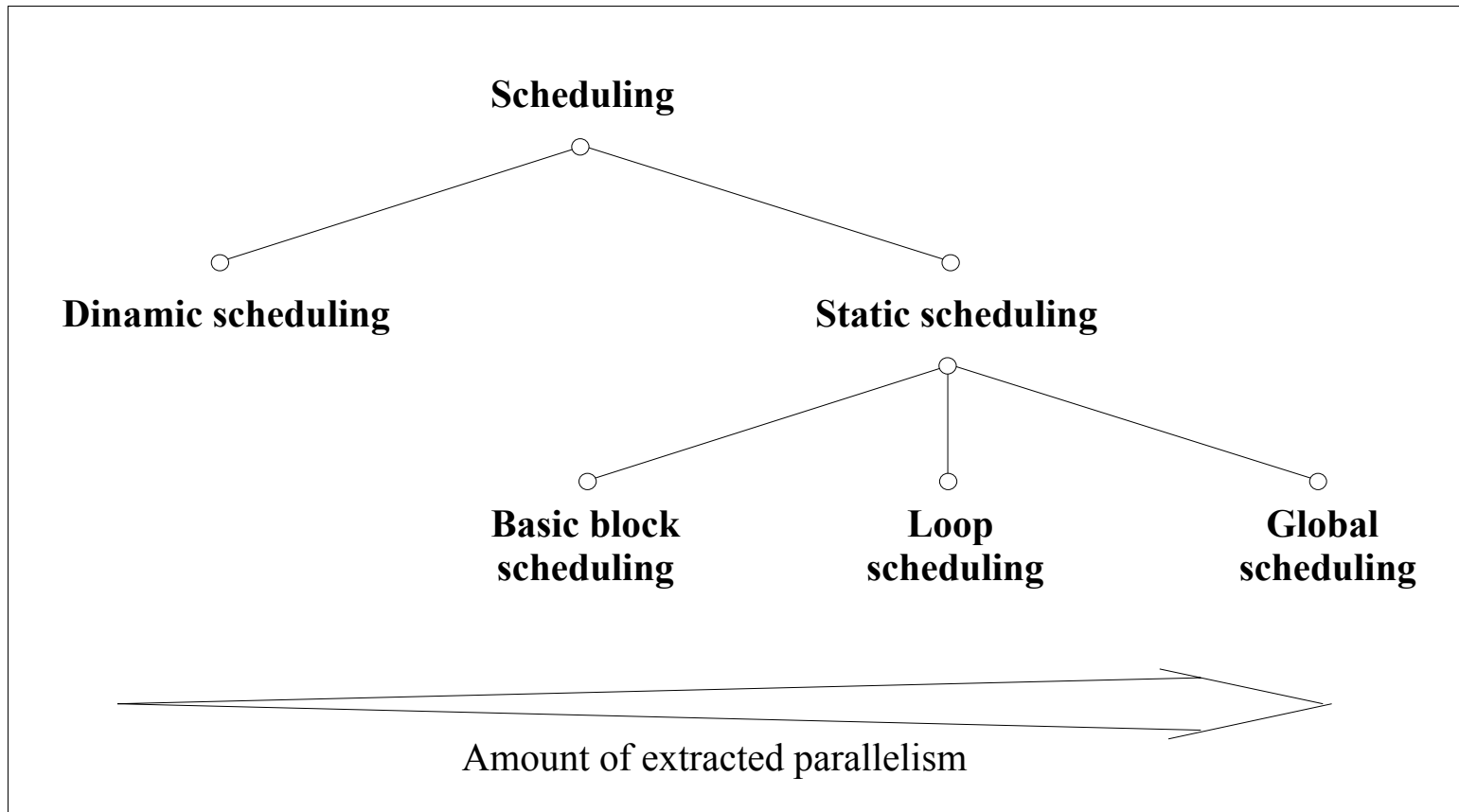


Extracción del paralelismo de instrucciones

❖ Planificación:

- Dinámica: Comprueba las dependencias y emite instrucciones independientes
- Estática: Comprueba **dependencias y optimiza el código** =>
 - ✓ Mayor capacidad para extraer el paralelismo existente en los programas

Extracción del paralelismo de instrucciones



Planificación de bloques básicos..

- ❖ Técnica sencilla. Eficiencia limitada por
 - El pequeño tamaño de los bloques básicos
 - ✓ Tamaño medio (en MIPS 3000):
 - 8 instrucciones en (7) programas de propósito general
 - 32 instrucciones en (3) programas de orientación científica
 - Las *verdaderas* dependencias de datos (RAW)
 - ✓ Las otras se pueden eliminar red denominando registros
- ❖ Ganancia en velocidad que proporciona
 - Menor que 2 en programas de propósito general
 - Menor que 4 en programas de orientación científica

Planificación de bloques básicos: grafo de dependencias de datos (DDG)

- ❖ Muestra las precedencias impuestas por ese tipo de dependencias
- ❖ A la derecha se tiene el grafo que corresponde al bloque básico

i1: ini: add r3, r1, r2

i2: sub r4, r1, r2

i3: mul r5, r3, r4

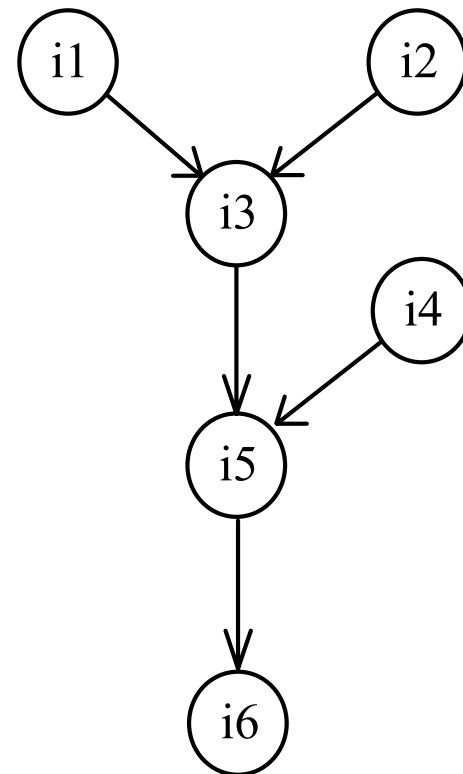
i4: mul r7, r6, r6

i5: sub r8, r7, r5

i6: jn eti

En forma texto:

{(i1,i3), (i2,i3), (i3,i5), (i4,i5), (i5,i6)}



Planificación de bloques básicos..

- ❖ La **construcción del DDG** es la **tarea** del planificador **que más tiempo consume**
 - n instrucciones $\Rightarrow n*(n-1)/2$ pares de instrucciones por analizar
- ❖ Tipos de planificadores de bloques básicos
 - Planificadores de listas (*list schedulers*)
 - ✓ Los más usados
 - Planificadores basados en criterios
 - ✓ Planifican, en primer lugar el primer elemento (la primera instrucción) que mejor cumpla una secuencia de criterios

Planificadores de listas

- ❖ Los elementos (instrucciones en nuestro caso) se planifican en pasos. En cada **paso**
 - 1º/ Regla de **selección** para crear una lista de elementos elegibles
 - 2º/ Regla para **elegir** de la lista anterior **el mejor** conjunto de elementos (una instrucción en nuestro caso) para planificar en este paso

Ejemplo de planificador de listas: algoritmo de Warren..

- ❖ En cada paso, sobre el grafo de dependencias de datos
 - Crear el conjunto de nodos elegibles. Es elegible
 - ✓ Todo nodo que no tienen predecesor
 - ✓ Aquel nodo cuyos predecesores no bloquean su ejecución porque ya han producido los datos y están disponibles
 - Esta condición vendrá dada por:
valor del contador de tiempo $\geq$
 $\geq$ tiempo más temprano de ejecución (v. trp. sgte.)

Ejemplo de planificador de listas: algoritmo de Warren..

- Elegir el que se va a planificar y quitarlo del grafo
 - ✓ El que está en el camino crítico
 - El camino más largo hasta el final del bloque básico
 - La longitud del camino se calcula sumando las latencias de ejecución de los arcos del camino
- Aumentar en 1 el contador de tiempo
- Calcular el **tiempo más temprano de ejecución** del nodo sucesor al que se ha quitado del grafo
 - ✓ Sumar el contador de tiempo (CT) y la latencia entre el nodo suprimido y su sucesor
 - En futuros pasos el nodo será **elegible si**
valor del CT $\geq$ tiempo más temprano de ejecución

Ejemplo de planificador de listas: algoritmo de Warren..

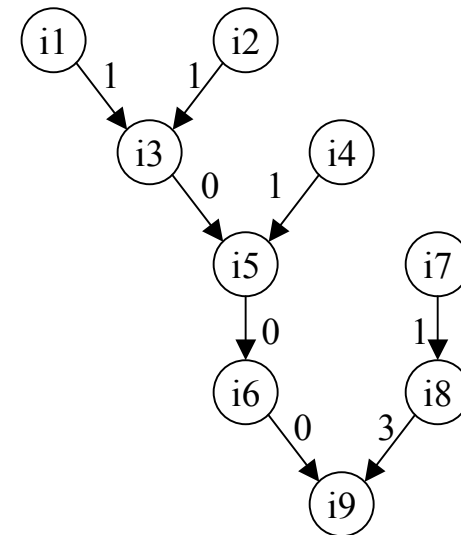
❖ Suponemos que al siguiente código

```
i1:      l      r10, b (r20)
i2:      l      r11, c (r20)
i3:      add    r12, r10, r11
i4:      l      r13, d (r20)
i5:      sub    r14, r12, r13
i6:      st     r14, a (r20)
i7:      l      r15, e (r20)
i8:      cmp    r16, r15, 0
i9:      bc     r16, ...
```

corresponde el **grafo de dependencias de datos** de la derecha

❖ En los arcos del grafo se indican las latencias

❖ Aunque no hay dependencia entre i6 e i9, i9 debe ser la última instrucción (si no no estaríamos en un bloque básico)

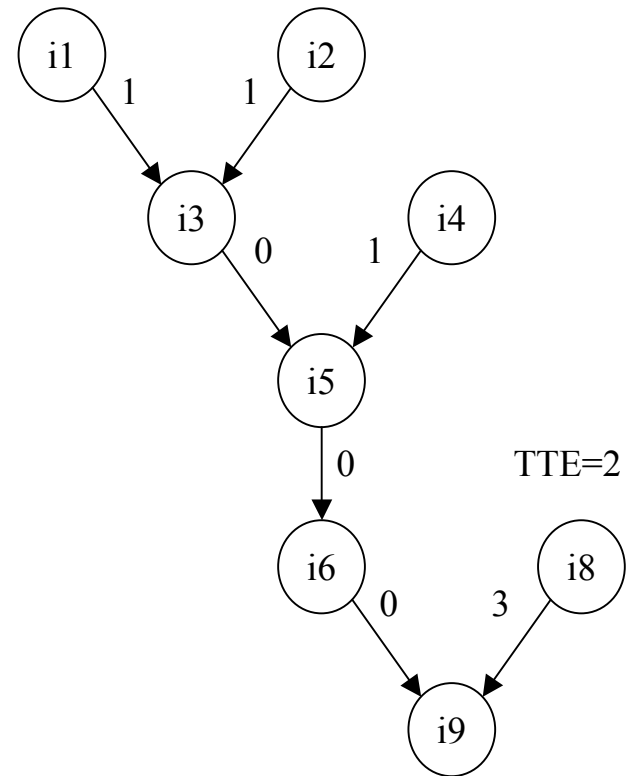


Ejemplo de planificador de listas: algoritmo de Warren..

❖ Contador de tiempo = 0

❖ Paso 1:

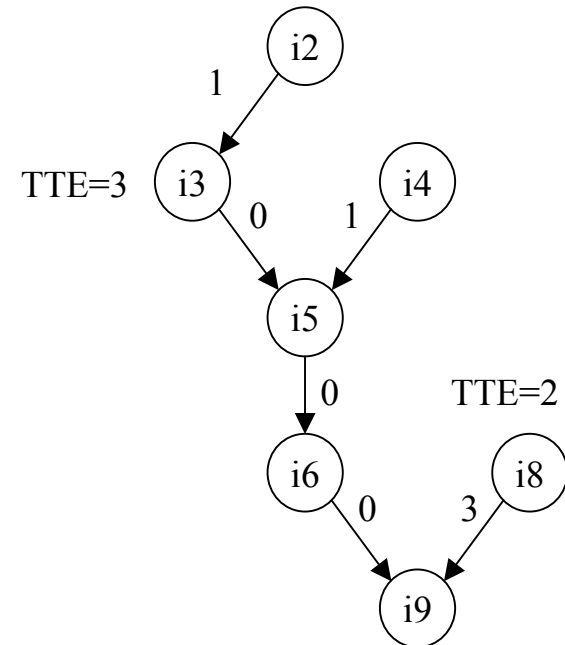
- Nodos elegibles: los que no tienen predecesor:
 $\{i1, i2, i4, i7\}$
- Longitudes respectivas caminos de estos nodos: 1, 1, 1, 4 $\Rightarrow$
Nodo planificado i7
- $CT = CT+1$ ($CT=1$)
- Tiempo más temprano de ejecución de i8 (sucesor de i7 en el grafo) = $CT+1 = 2$



Ejemplo de planificador de listas: algoritmo de Warren..

❖ Paso 2:

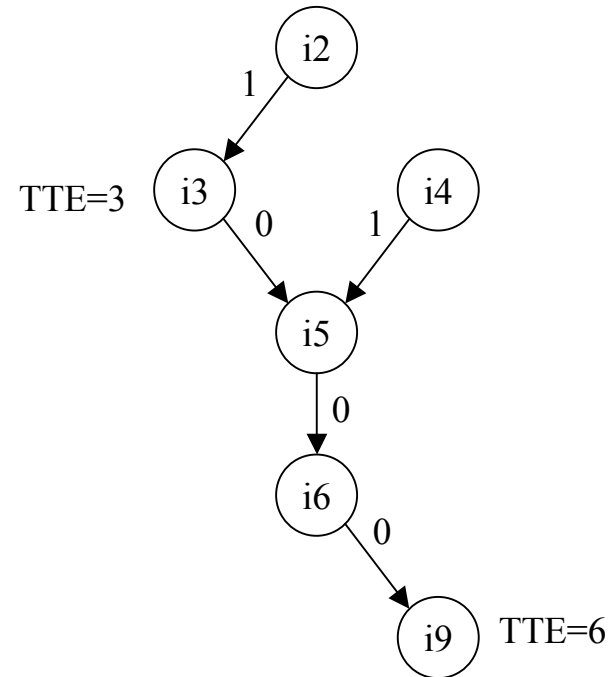
- Nodos elegibles {i1, i2, i4}
(i8 no es elegible porque $CT=1 < TTE(i8)=2$)
- Longitudes respectivas caminos:
1, 1, 1 $\Rightarrow$ **Nodo planificado i1**
(se deshace el empate
escogiendo el primero)
- $CT = 2$
- $TTE(i3) = CT + \text{latencia}(i1, i3) = 2 + 1 = 3$



Ejemplo de planificador de listas: algoritmo de Warren..

❖ Paso 3:

- Nodos elegibles {i2, i4, i8}
($CT=2 \geq TTE(i8)=2$)
- Longitudes respectivas
caminos: 1, 1, 3 => **Nodo
planificado i8**
- $CT = 3$
- $TTE(i9) = CT + \text{latencia}(i8, i9) =$
 $= 3 + 3 = 6$



Ejemplo de planificador de listas: algoritmo de Warren

- ❖ En pasos sucesivos se planifican, por este orden: i2, i4, i3, i5, i6, i9. Por tanto, el bloque básico planificado queda

i7:	l	r15, e (r20)
i1:	l	r10, b (r20)
i8:	cmp	r16, r15, 0
i2:	l	r11, c (r20)
i4:	l	r13, d (r20)
i3:	add	r12, r10, r11
i5:	sub	r14, r12, r13
i6:	st	r14, a (r20)
i9:	bc	r16, ...

- ❖ Este bloque consume 9 ciclos y el original 15. Así pues se ha producido una mejora de $15/9 = 1,67$

Planificación de bloques básicos

- ❖ Los planificadores de bloques básicos son usados por procesadores encauzados y por los primeros procesadores superescalares
 - Los VLIW no los utilizan porque necesitan una planificación más ambiciosa, es decir:
 - ✓ Planificación de bucles y
 - ✓ Planificación global

Planificación de bucles

- ❖ Los bucles son la principal fuente de paralelismo de instrucciones
- ❖ Dos técnicas fundamentales para planificar bucles
 - Desenrollamiento de bucles
 - ✓ Interviene como componente de métodos de planificación más sofisticados
 - Encauzamiento software y planificación de trazas
 - Encauzamiento software (*software pipelining*)
 - ✓ Técnica estándar de planificación de bucles

Encauzamiento software..

❖ Tomemos

- Cuerpo del bucle

fload *f10, 0 (r2)*
fmul *f10, f2, f10*
fstore *200(r2), f10*
addi *r2, r2, 1*

- Procesador ILP:

- ✓ Unidades funcionales separadas para **FP**, **FX** (operaciones enteras), *load* y *store*
 - En todas el intervalo de iniciación es 1 ciclo
- ✓ Latencias por dependencias de datos: *fmul*, **dos ciclos**; el resto cero ciclos

❖ Ejecución de una iteración

Ciclo	Instrucción
c	<i>fload</i> <i>f10, 0(r2)</i>
c+1	<i>fmul</i> <i>f10, f2, f10</i>
c+2	<i>addi</i> <i>r2, r2, 1</i>
c+3	<i>nop</i>
c+4	<i>fstore</i> <i>199(r2), f10</i>

❖ La siguiente iteración puede empezar en el ciclo c+1

c+1	<i>fload</i> <i>f11, 0(r2)</i>
c+2	<i>fmul</i> <i>f11, f2, f11</i>
c+3	<i>addi</i> <i>r2, r2, 1</i>
c+4	<i>nop</i>
c+5	<i>fstore</i> <i>199(r2), f11</i>

Encauzamiento software..

❖ El resultado para 7 pasadas será

Ciclo	Iteración						
	1	2	3	4	5	6	7
c	fload						
c+1	fmult	fload					
c+2	addi	fmult	fload				
c+3	nop	addi	fmult	fload			
c+4	fstore	nop	addi	fmult	fload		
c+5		fstore	nop	addi	fmult	fload	
c+6			fstore	nop	addi	fmult	fload
c+7				fstore	nop	addi	fmult
c+8					fstore	nop	addi
c+9						fstore	nop
c+10							fstore

Encauzamiento software..

- ❖ Los ciclos $c+4$, $c+5$ y $c+6$:
 - Utilizan todo el paralelismo disponible
 - Siguen un patrón repetitivo $\Rightarrow$ Pueden ser reemplazados por un bucle
- ❖ El código del encauzamiento software queda dividido en tres partes
 - Código de inicio (*start-up code*, *prologue* o *prelude*)
 - Bucle del patrón repetitivo
 - Código de finalización (*epilogue* o *postlude*)

Encauzamiento software

❖ Problema principal:

- Encontrar el patrón repetitivo que pueda ejecutarse en el menor número de ciclos
 - ✓ No siempre es tan sencillo como en el ejemplo anterior

❖ Un desenrollamiento previo del bucle suele ser fundamental para un encauzamiento software más efectivo

Planificación global

- ❖ Planificación más allá de los límites de los bloques básicos y de los bucles
- ❖ Tareas propias de la planificación global
 - Planificación con **movimiento de código atravesando los límites de los bloques básicos**
 - Planificación de bucles
 - Manejar llamadas a procedimientos dentro de bucles, manejar dependencias de control, dar soporte a la política de emisión de instrucciones (típico de procesadores superescalares), etc.

Técnica ejemplo de planificación global: planificación de trazas

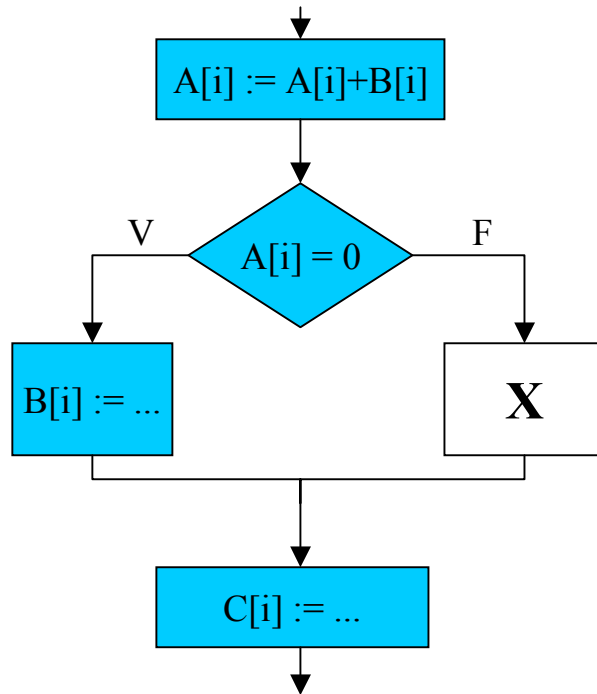
❖ 1º: **selección** de trazas

- Traza: posible secuencia de ejecución de operaciones dentro de un bucle (puede incluir bifurcaciones y saltos)
- Se selecciona la traza **más probable**; después, la siguiente más probable, etc. hasta que toda operación del programa pertenezca a alguna traza seleccionada

❖ 2º: **compactación** de trazas

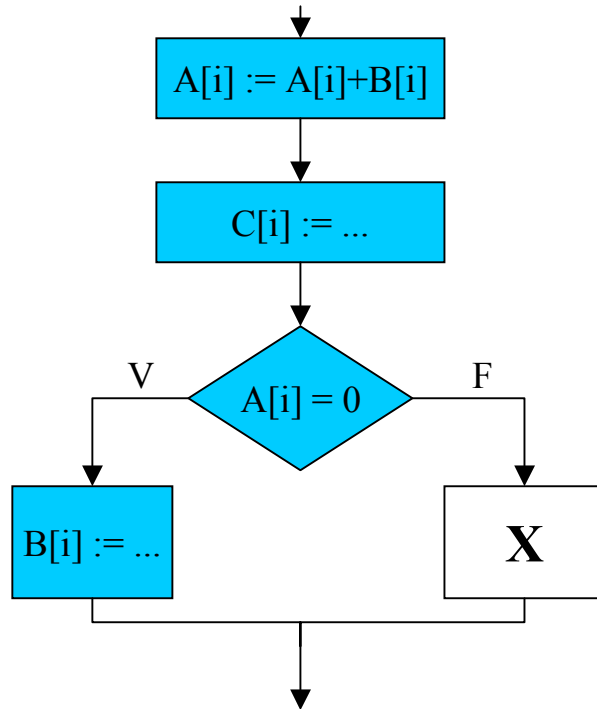
- Las operaciones de cada traza se paralelizan agrupándose en un número menor de instrucciones grandes (VLIW)

Ejemplo de selección de una traza



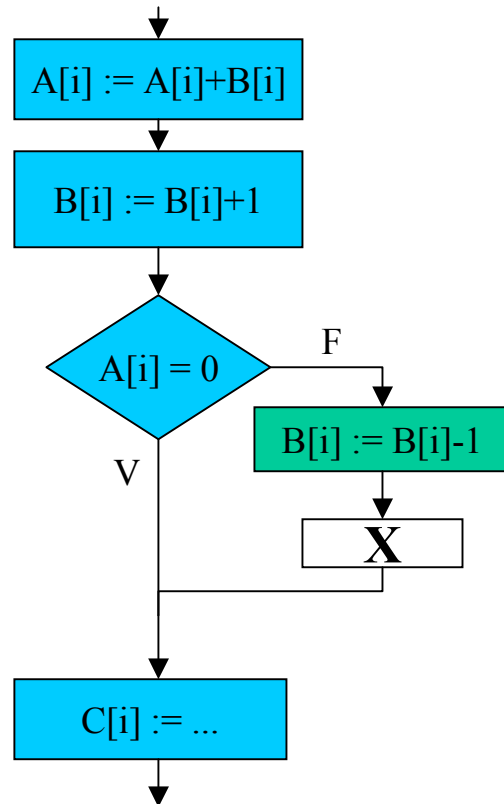
- ❖ Se ha seleccionado la traza con fondo azul porque se prevee más probable que la condición sea verdadera
- ❖ Una vez seleccionada hay que compactar la traza
 - La compactación supone **mover código hacia arriba**

Movimiento de código en planificación de trazas



❖ Si “ $C[i] := \dots$ ”
no tiene
dependencias,
puede colocarse
antes de la
bifurcación

Movimiento de código en planificación de trazas



❖ Si “ $B[i] := \dots$ ” no tiene dependencias:

- Se puede colocar antes de la bifurcación
 - ✓ Movimiento **especulativo** de código: sólo útil si previsión acertada
- Suele ser preciso **añadir código compensatorio** ($B[i] := B[i] - 1$) por si se sigue el camino no previsto (*bookkeeping code*)

Planificación de trazas: principales dificultades

- ❖ El **obstáculo más importante** para mover instrucciones son las **dependencias de datos RAW**
- ❖ Los **movimientos especulativos** de código son **problemáticos**
 - Provocan una considerable **expansión de código**
 - Es clave para la mejora del rendimiento que las trazas previstas correspondan a los caminos que sigue el programa un **alto porcentaje** de veces
 - ✓ En los caminos no previstos el rendimiento se degrada por la ejecución del código compensatorio
 - Surgen dificultades cuando el código que se mueve puede provocar **interrupción** (el código compensatorio no las resuelve)
- ❖ Cuando no hay suficiente **paralelismo en las trazas** un **desenrollamiento de bucles** puede suministrar ese paralelismo