Optimización de Código

CI4722 – Lenguajes de Programación III

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"

UNIVERSIDAD SIMÓN BOLÍVAR

Optimización de Código

. . . o al menos un buen intento

- Optimización de Código – eufemismo por "mejorarlo tanto como se pueda sin cambiar la semántica"
- Generar código razonablemente bueno es el paso intermedio – mejorar el código en la plataforma específica es el último paso.
- La traducción directa de alto nivel a lenguaje intermedio generalmente introduce complejidad adicional en la ejecución.
- Mejorar la calidad del código intermedio es el paso inicial.

UNIVERSIDAD SIMÓN BOLÍVAR

¿Dónde enfocar el esfuerzo de mejora?

Focus, Daniel san!

- **Optimizaciones Locales**
  - Aplican sobre un bloque básico aislado del resto.
  - Las introdujimos en nuestro estudio de generación de código.
  - Aplicables tanto a código intermedio como a código final y *quieres* tenerlas en ambos niveles.

UNIVERSIDAD SIMÓN BOLÍVAR

¿Dónde enfocar el esfuerzo de mejora?

Focus, Daniel san!

- **Optimizaciones Locales**
  - Aplican sobre un bloque básico aislado del resto.
  - Las introdujimos en nuestro estudio de generación de código.
  - Aplicables tanto a código intermedio como a código final y *quieres* tenerlas en ambos niveles.
- **Optimizaciones Globales**
  - Aplican sobre todo el programa, explotando las relaciones entre varios bloques básicos simultáneamente.
  - Son dirigidas por **Análisis de Flujo de Datos** *(Data Flow Analysis)*.
  - Aplicables tanto a código intermedio como a código final – en general se aplican **agresivamente** al intermedio.

UNIVERSIDAD SIMÓN BOLÍVAR

Optimizaciones más frecuentes

Combinación de constantes
(Constant Folding and Combining)

- Eliminar cómputos con valores constantes a tiempo de compilación.
- El *front-end* puede detectar los de alto nivel –
  generación intermedia podría introducir cómputos constantes.
- Aplicable localmente.

UNIVERSIDAD SIMÓN BOLÍVAR

Combinación de Constantes – *Folding*

- *Folding* – cuando hay *una* operación con constantes
- Identificar operaciones con constantes en ambos operadores.
- Sustituir por la asignación con el resultado de la operación.

```
t1 := 21 * 2
```

$\Rightarrow$

```
t1 := 42
```

UNIVERSIDAD SIMÓN BOLÍVAR

Combinación de Constantes – *Combining*

- *Combining* – cuando hay *varias* operaciones con constantes
- Dos operaciones I_1 e I_2 en el *mismo* bloque.
- Ambas tienen al menos un operador constante.
- I_2 usa al resultado de I_1.

```
t2 := 2 * t1
...
t3 := t2 * 21
```

$\Rightarrow$

```
t2 := 2 * t1
...
t3 := 42 * t1
```

UNIVERSIDAD SIMÓN BOLÍVAR

Optimizaciones más frecuentes

Reducción de Fuerza
(Strength Reduction)

- Reemplazar operaciones costosas por alternativas económicas.
- Multiplicación por dos, reemplazarla por suma.
- Multiplicaciones por potencias de dos, reemplazarlas por *shift*.
- Aplicable localmente.

UNIVERSIDAD SIMÓN BOLÍVAR

Optimizaciones más frecuentes

Propagación de Constantes
(Constant Propagation)

- Reemplazar uso de registros por constantes.
- Operación *I* mueve una constante manifiesta a un registro – ocurrencias posteriores del registro se reemplazan por la constante.

```
t1 := 5
t2 := x
t3 := 7
t4 := t4 + t1
t1 := t1 + t2
t1 := t1 + 1
t3 := 12
t8 := t1 - t2
t9 := t3 + t5
t3 := t2 + 1
```

$\Rightarrow$

```
t1 := 5
t2 := x
t3 := 7
t4 := t4 + 5
t1 := 5  + x
t1 := 6  + x
t3 := 12
t8 := 6
t9 := 12 + t5
t3 := x  + 1
```

UNIVERSIDAD SIMÓN BOLÍVAR

Optimizaciones más frecuentes

Eliminación de Subexpresiones Comunes
(Common Subexpression Elimination)

- Reemplazar una operación por una copia.
- Operaciones I_1 e I_2 con lado derecho idéntico – I_2 es posterior a I_1.
- Los operandos de I_1 no son redefinidos entre I_1 e I_2.
- Aprovechar el destino de I_1 en el lado derecho de I_2.
- Aplicable localmente.

```
t2 := t1 - 1
t5 := t4 + 1
t7 := t2 + t3
t5 := t1 - 1
```

$\Rightarrow$

```
t2 := t1 - 1
t5 := t4 + 1
t7 := t2 + t3
t5 := t2
```

UNIVERSIDAD SIMÓN BOLÍVAR

Optimizaciones más frecuentes

Retroceso de Copias
(Backward Copy Propagation)

- Eliminar copias redundantes.
- Operaciones I_1 e I_2 en el mismo bloque – I_2 es una copia.
- I_2 usa el destino de I_1, que **no** está vivo al salir del bloque.
- Aplicable localmente.

Suponiendo que `t6` no está viva al salir

```
t2 := t9 + t1
t4 := t2
t6 := t2 + 1
t9 := t1
t7 := t6
t5 := t7 + 1
t8 := t2 + t7
```

$\Rightarrow$

```
t2 := t9 + t1
t4 := t2
t7 := t2 + 1
t9 := t1
t5 := t7 + 1
t8 := t2 + t7
```

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo eliminar la redundancia globalmente?

```
void quicksort(int m,int n) {
  int i, j, v, x;
  if (n <= m) return;
  /* inicio */
  i = m-1; j = n; v = a[n];
  while (1) {
    do i = i+1; while (a[i] < v);
    do j = j-1; while (a[j] > v);
    if (i >= j) break;
    x = a[i]; a[i] = a[j]; a[j] = x;
  }
  x = a[i]; a[i] = a[n]; a[n] = x;
  /* final */
  quicksort(m,j); quicksort(i+1,n);
}
```

Quicksort “en sitio”

UNIVERSIDAD SIMÓN BOLÍVAR

Suposiciones

- Solamente nos interesa el fragmento de código marcado
  - Incluye ciclos anidados.
  - Incluye acceso a arreglos.
- El tipo entero ocupa 4 bytes.
- El acceso `x := a[i]` se traduce

```
t := 4 * i
x := a[t]
```

- La asignación `a[j] := x` se traduce

```
t    := 4 * i
a[t] := x
```

Esto va a producir *mucha* redundancia...

UNIVERSIDAD SIMÓN BOLÍVAR

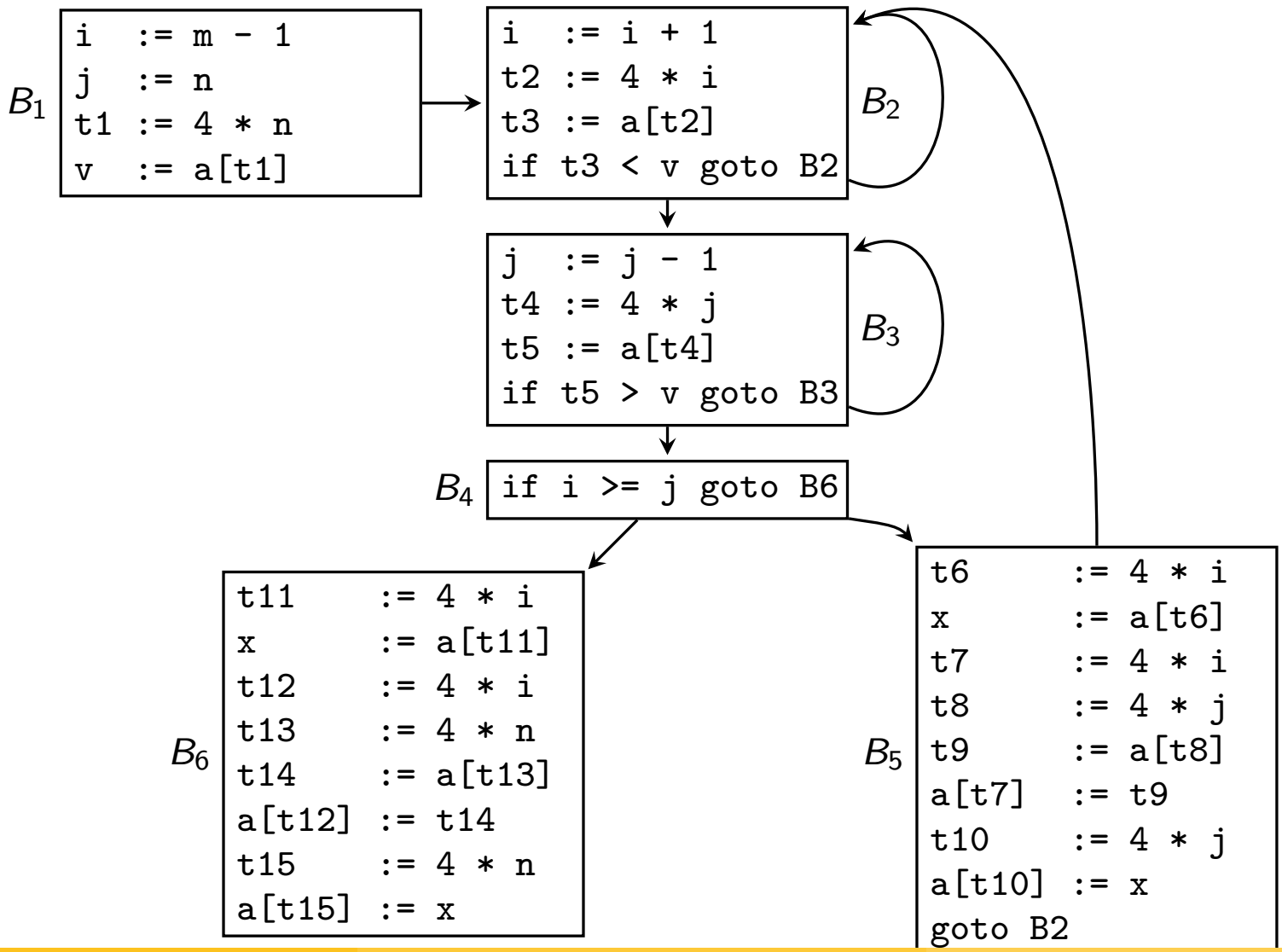
Se compila a código intermedio

```
 (1) i   := m - 1
 (2) j   := n
 (3) t1  := 4 * n
 (4) v   := a[t1]
 (5) i   := i + 1
 (6) t2  := 4 * i
 (7) t3  := a[t2]
 (8) if t3 < v goto (5)
 (9) j   := j - 1
(10) t4  := 4 * j
(11) t5  := a[t4]
(12) if t5 > v goto (9)
(13) if i >= j goto (23)
(14) t6  := 4 * i
(15) x   := a[t6]
```

```
(16) t7      := 4 * i
(17) t8      := 4 * j
(18) t9      := a[t8]
(19) a[t7]   := t9
(20) t10     := 4 * j
(21) a[t10]  := x
(22) goto (5)
(23) t11     := 4 * i
(24) x       := a[t11]
(25) t12     := 4 * i
(26) t13     := 4 * n
(27) t14     := a[t13]
(28) a[t12]  := t14
(29) t15     := 4 * n
(30) a[t15]  := x
```

UNIVERSIDAD SIMÓN BOLÍVAR

Y construimos el grafo de flujo



Eliminación de Subexpresiones Comunes

- El acceso a arreglos introduce instrucciones redundantes – el programador no puede evitarlas, el compilador si.
- En el bloque B_5 se repiten los cálculos para `4 * i` y `4 * j`

```
t6      := 4 * i
x       := a[t6]
t7      := 4 * i
t8      := 4 * j
t9      := a[t8]
a[t7]   := t9
t10     := 4 * j
a[t10]  := x
goto B2
```

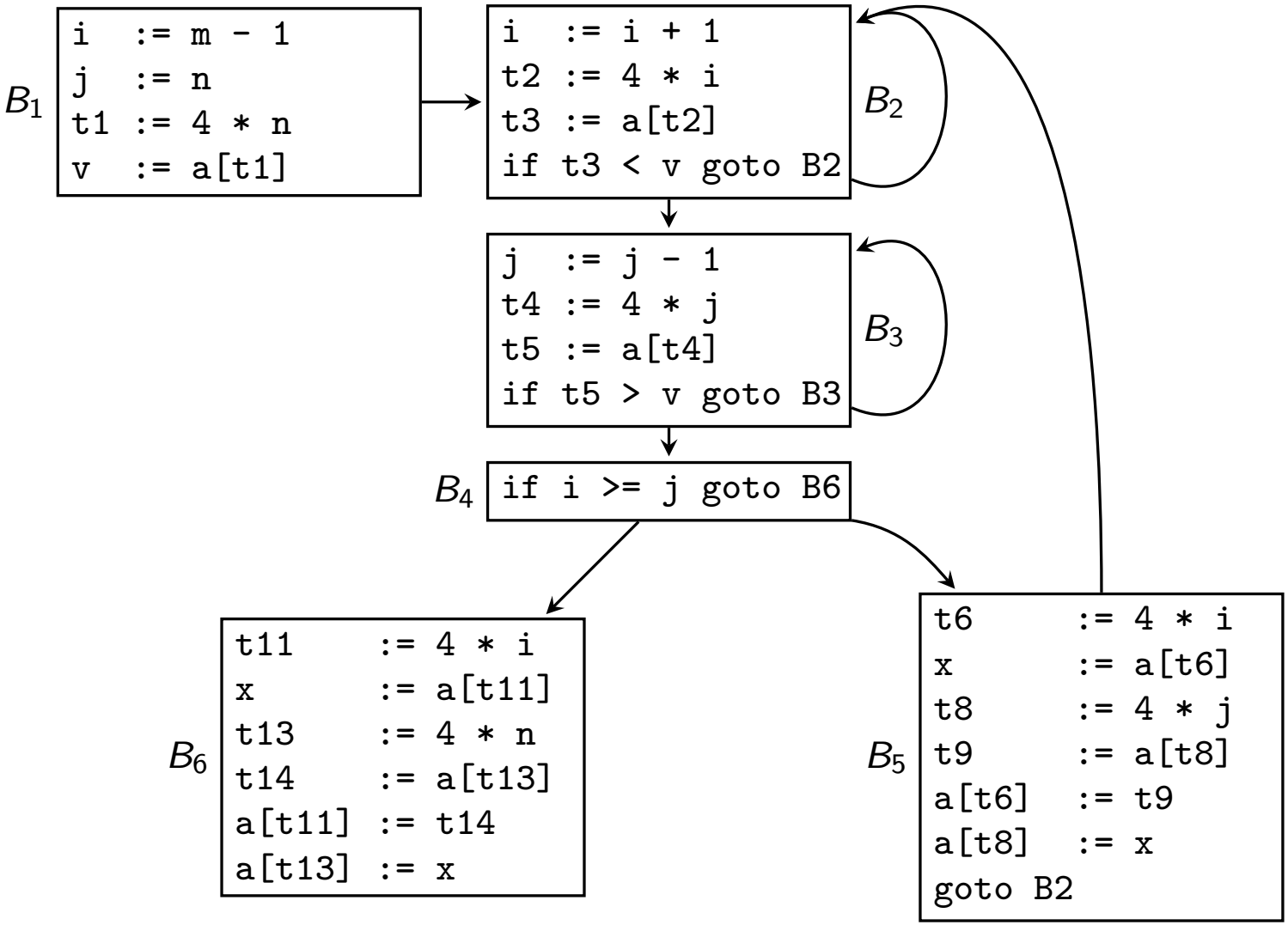
$\Longrightarrow$

```
t6     := 4 * i
x      := a[t6]
t8     := 4 * j
t9     := a[t8]
a[t6]  := t9
a[t8]  := x
goto B2
```

- En el bloque B_6 se repiten los cálculos para `4 * i` y `4 * n`

UNIVERSIDAD SIMÓN BOLÍVAR

Eliminación de Subexpresiones Comunes



UNIVERSIDAD SIMÓN BOLÍVAR

Eliminación de Subexpresiones Comunes

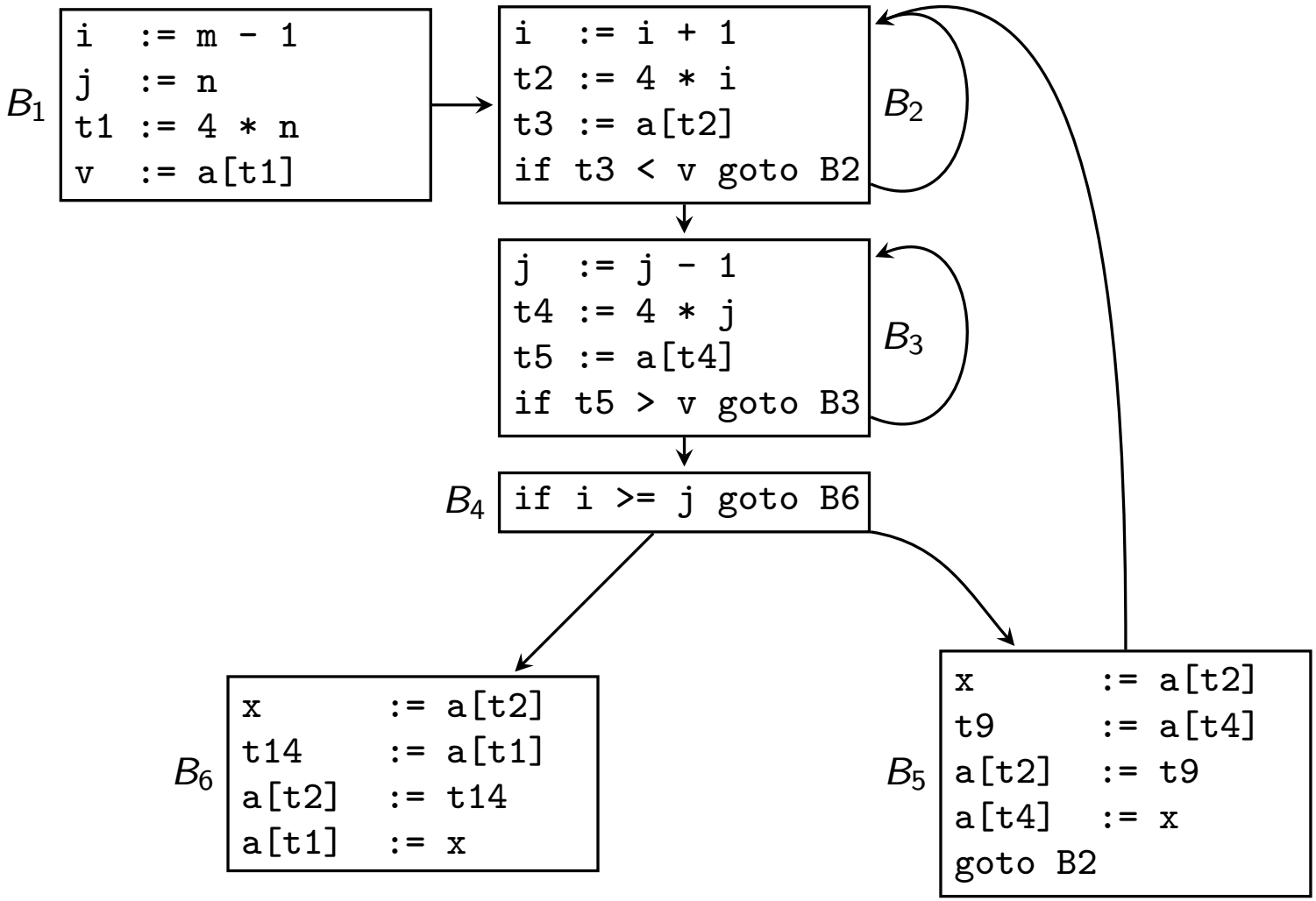
Observemos *globalmente*

- B_5 aún evalúa `4 * i` y `4 * j` – una sola vez.
- B_6 aún evalúa `4 * i` y `4 * n` – una sola vez.
- B_3 evalúa `4 * j` en `t4` – ¡pero `t4` no cambia entre B_3 y B_5!
- B_2 evalúa `4 * i` en `t2` – ¡pero `t2` no cambia entre B_2 y B_5 o B_6!
- B_1 evalúa `4 * n` en `t1` – ¡pero `t1` no cambia entre B_1 y B_6!

Lo que nos lleva a...

UNIVERSIDAD SIMÓN BOLÍVAR

Eliminación de Subexpresiones Comunes



UNIVERSIDAD SIMÓN BOLÍVAR

Eliminación de Subexpresiones Comunes

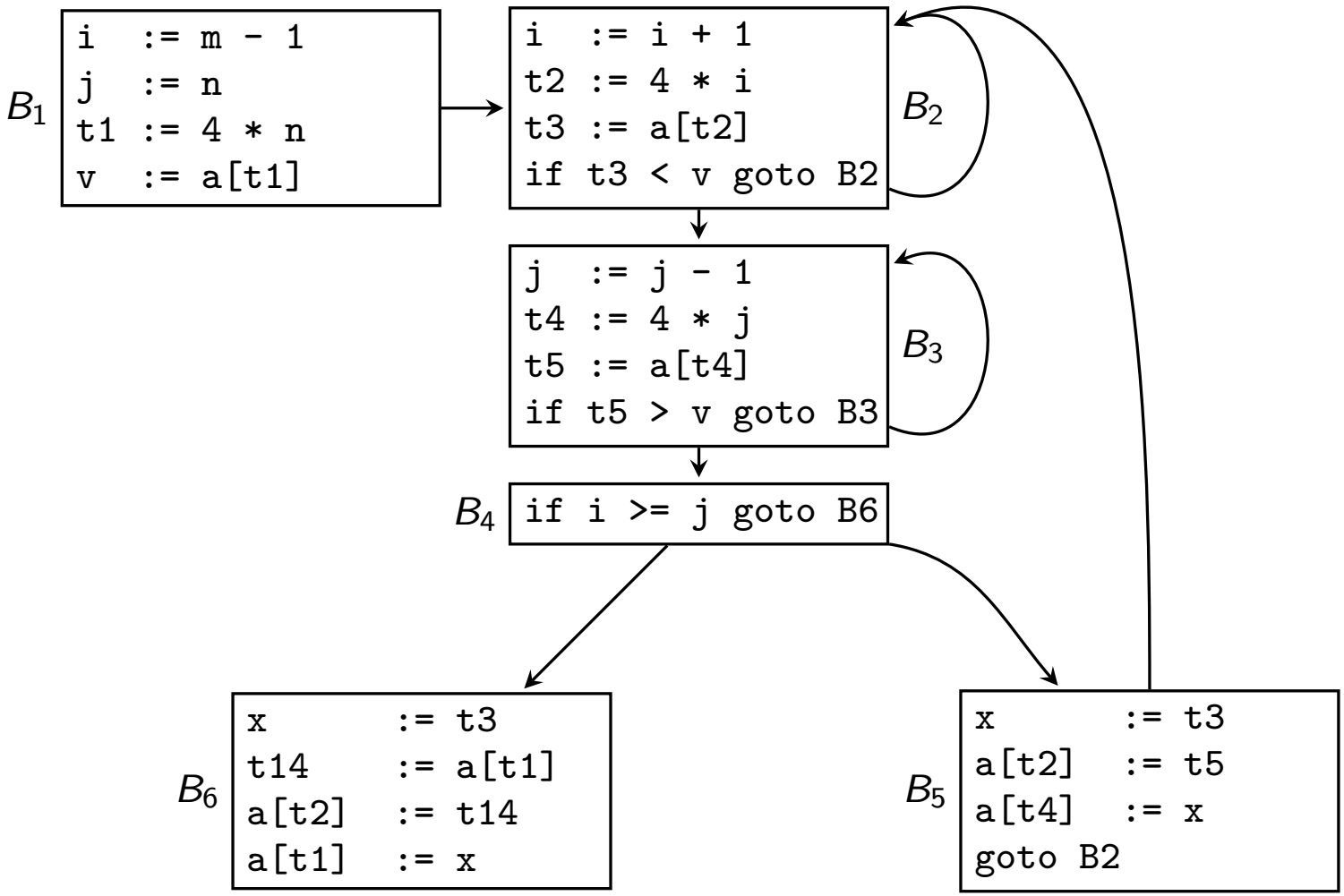
Observemos *globalmente*

- B_2 evalúa `a[t2]` en `t3` – ¡pero `t3` no cambia entre B_2 y B_5 o B_6!
- B_3 evalúa `a[t4]` en `t5` – ¡pero `t5` no cambia entre B_3 y B_5!
- En B_5 eliminamos la copia inútil.

Lo que nos lleva a...

UNIVERSIDAD SIMÓN BOLÍVAR

Eliminación de Subexpresiones Comunes



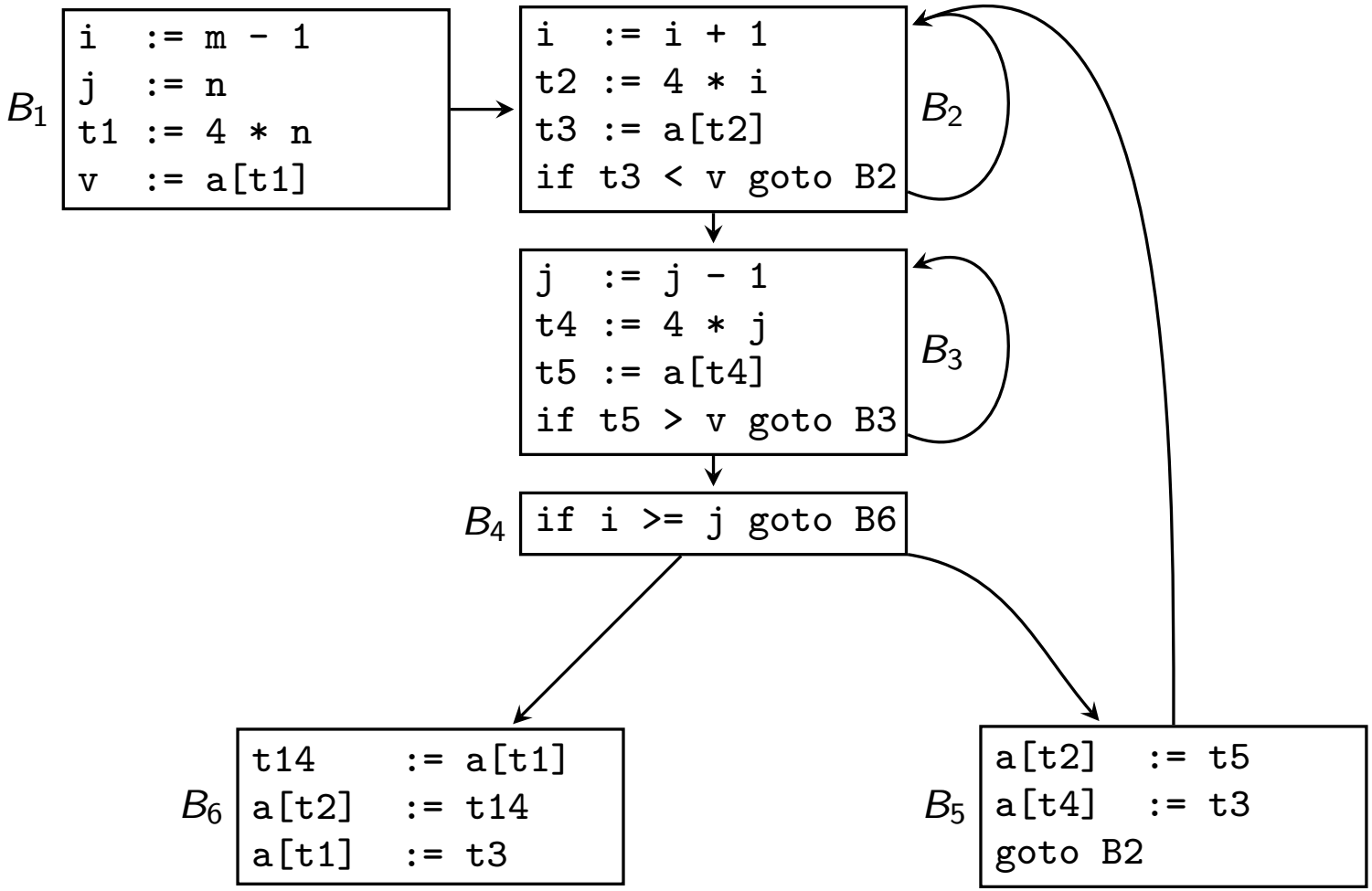
UNIVERSIDAD SIMÓN BOLÍVAR

Propagación de Copias / Código Muerto

- En general, buscar `u := v` y tratar de usar `v` en lugar de `u` en instrucciones que le sigan.
- En B_5 tenemos `x := t3` así que podemos reutilizar `t3` en el mismo bloque – no parece mucha mejora a simple vista.
- Pero ahora, la asignación `x := t3` es absolutamente inútil – código muerto que podemos eliminar.

UNIVERSIDAD SIMÓN BOLÍVAR

Propagación de Copias / Código Muerto



Mejorando los ciclos

Expresiones invariantes

- Los programas pasan la mayor parte del tiempo en ciclos – mientras menos instrucciones tengan, más rápido ejecutan.
- Cualquier expresión cuyo valor es *independiente* de la cantidad de veces que se ejecute el ciclo es una *invariante del ciclo*.
  - No tiene sentido evaluarla dentro del ciclo.
  - Se mueve el código (*Code Motion*) fuera del ciclo.
- El destino del código es el bloque de entrada al ciclo.

UNIVERSIDAD SIMÓN BOLÍVAR

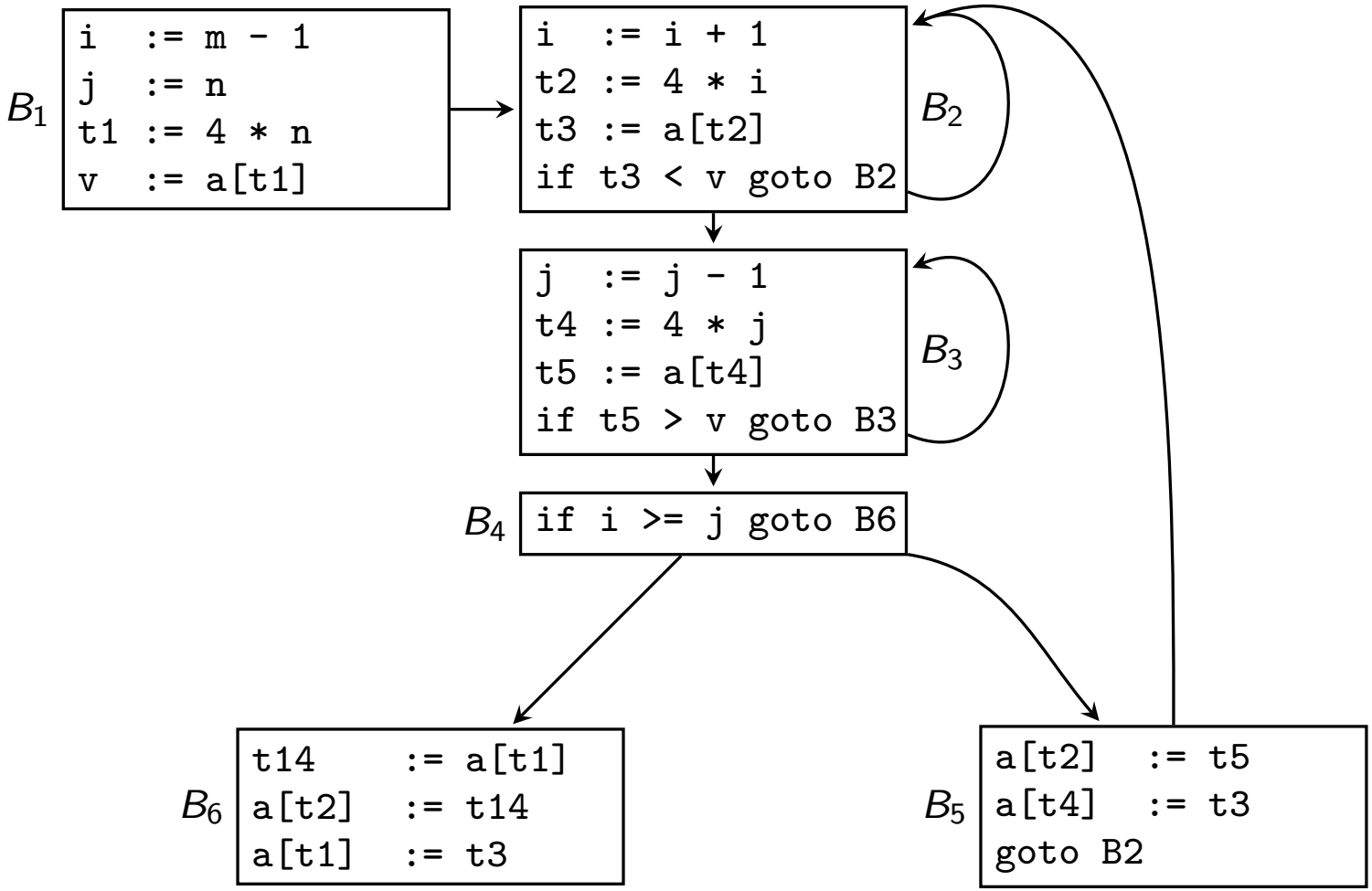
Mejorando los ciclos

Variables de inducción

- Decimos que `i` es una **variable de inducción** para un ciclo L si existe una constante `c` tal que cada vez que se asigna un valor a `i`, su valor aumenta `c` – note que `c` puede ser positiva o negativa.
- Identificar las variables de inducción permite aplicar dos optimizaciones:
  - Transformar su cálculo a *una* suma o resta por iteración – un caso de Reducción de Fuerza muy rendidor.
  - Es posible expresar *varias* variables de inducción en función de una sola – "variables en cadencia"
- El análisis se hace desde los ciclos internos hacia los externos.

UNIVERSIDAD SIMÓN BOLÍVAR

Variables de Inducción



Variables de Inducción

- Consideremos el ciclo B_3
  - `j` y `t4` están en cadencia –
    cada vez que `j` disminuye en 1, `t4` disminuye en 4.
  - `j` es necesaria en B_4 y `t4` es necesaria en B_5 –
    no podemos eliminarlas del todo. . . todavía.
- Podemos aplicar Reducción de Fuerza
  - Podemos cambiar `t4 := 4 * j` por `t4 := t4 - 4`.
  - Falta darle valor inicial a `t4` *antes* de entrar al ciclo por primera vez.
  - *Agregamos* la inicialización en B_1 – justo donde se inicializa `j`.

UNIVERSIDAD SIMÓN BOLÍVAR

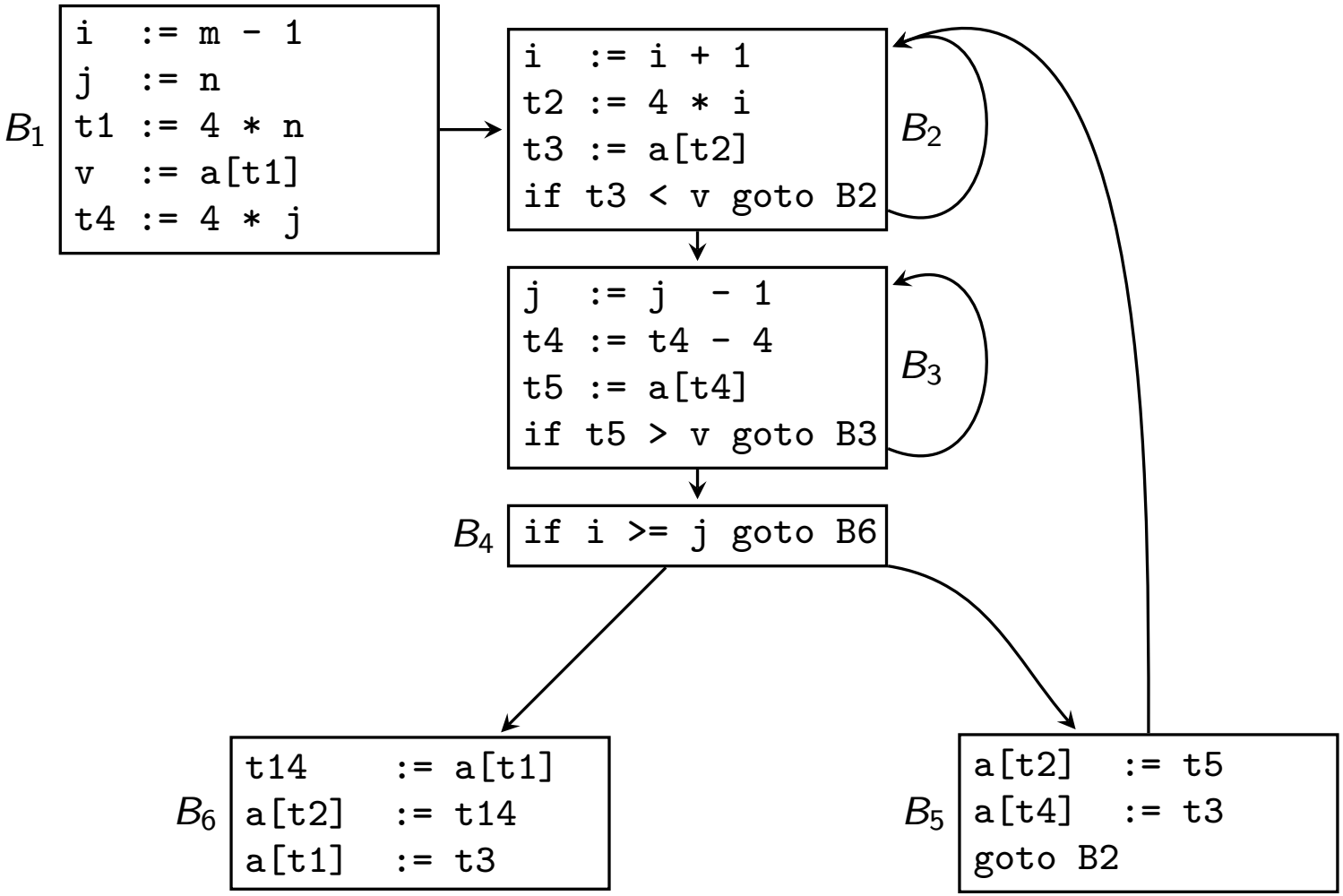
Variables de Inducción

- Consideremos el ciclo B_3
  - `j` y `t4` están en cadencia –
    cada vez que `j` disminuye en 1, `t4` disminuye en 4.
  - `j` es necesaria en B_4 y `t4` es necesaria en B_5 –
    no podemos eliminarlas del todo. . . todavía.
- Podemos aplicar Reducción de Fuerza
  - Podemos cambiar `t4 := 4 * j` por `t4 := t4 - 4`.
  - Falta darle valor inicial a `t4` *antes* de entrar al ciclo por primera vez.
  - *Agregamos* la inicialización en B_1 – justo donde se inicializa `j`.

Agregamos *una* multiplicación,
pero sustituimos *varias* por restas.

UNIVERSIDAD SIMÓN BOLÍVAR

Variables de Inducción



UNIVERSIDAD SIMÓN BOLÍVAR

Variables de Inducción

- Consideremos el ciclo B_2
  - `i` y `t2` están en cadencia –
    cada vez que `i` aumenta en 1, `t2` aumenta en 4.
  - `i` es necesaria en B_4 y `t2` es necesaria en B_5 y B_6 –
    no podemos eliminarlas del todo. . . todavía.
- Podemos aplicar Reducción de Fuerza
  - Podemos cambiar `t2 := 4 * i` por `t2 := t2 + 4`.
  - Falta darle valor inicial a `t2` *antes* de entrar al ciclo por primera vez.
  - *Agregamos* la inicialización en B_1 – justo donde se inicializa `i`.

UNIVERSIDAD SIMÓN BOLÍVAR

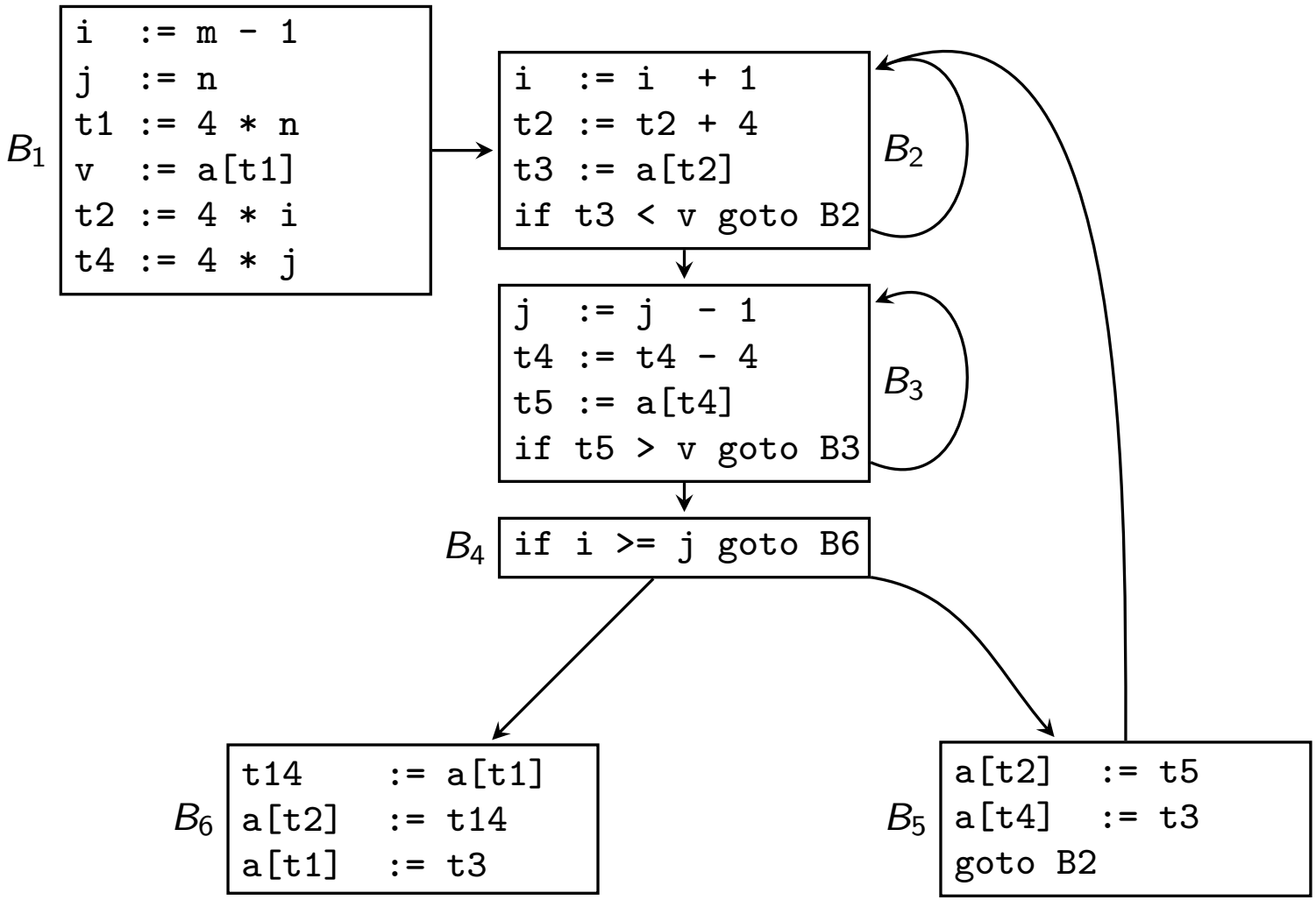
Variables de Inducción

- Consideremos el ciclo B_2
  - `i` y `t2` están en cadencia –
    cada vez que `i` aumenta en 1, `t2` aumenta en 4.
  - `i` es necesaria en B_4 y `t2` es necesaria en B_5 y B_6 –
    no podemos eliminarlas del todo. . . todavía.
- Podemos aplicar Reducción de Fuerza
  - Podemos cambiar `t2 := 4 * i` por `t2 := t2 + 4`.
  - Falta darle valor inicial a `t2` *antes* de entrar al ciclo por primera vez.
  - *Agregamos* la inicialización en B_1 – justo donde se inicializa `i`.

Agregamos *una* multiplicación,
pero sustituimos *varias* por sumas.

UNIVERSIDAD SIMÓN BOLÍVAR

Variables de Inducción



UNIVERSIDAD SIMÓN BOLÍVAR

Variables de Inducción

El golpe de gracia

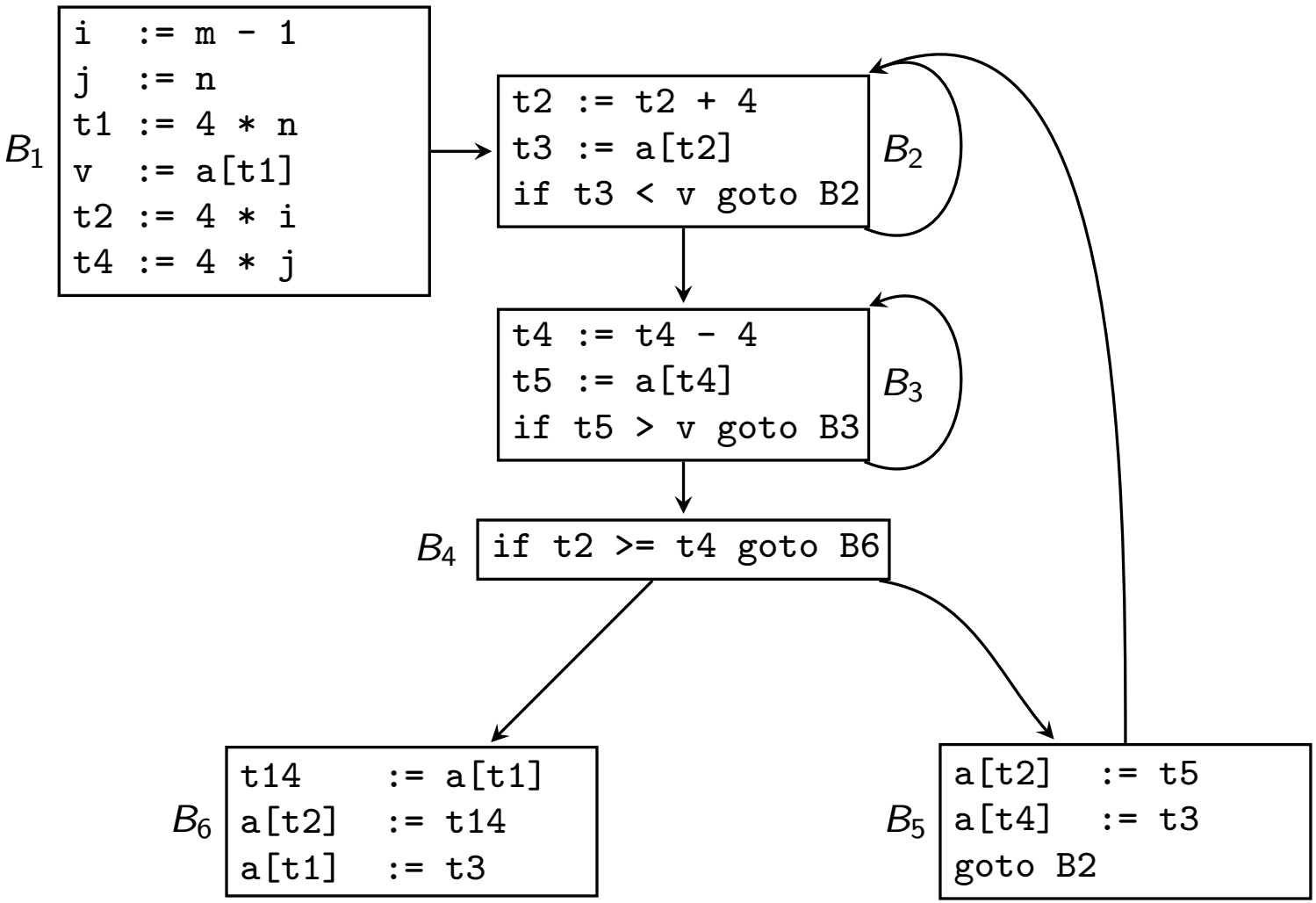
- Falta considerar el lazo $B_2 - B_3 - B_4 - B_5$
- El único uso de `i` y `j` es evaluar la condición en B_4.
- En virtud de los cambios de Reducción de Fuerza sabemos que

$$t2 = 4 \times i$$
$$t4 = 4 \times j$$

- ¡Podemos usar `t2 >= t4` en lugar de `i >= j`!
- Como consecuencia, las asignaciones a `i` y `j` se convierten en código muerto que puede ser eliminado.

UNIVERSIDAD SIMÓN BOLÍVAR

Variables de Inducción



UNIVERSIDAD SIMÓN BOLÍVAR

El código mejorado

```
 (1) i      := m - 1
 (2) j      := n
 (3) t1     := 4 * n
 (4) v      := a[t1]
 (5) t2     := 4 * i
 (6) t4     := 4 * j
 (7) t2     := t2 + 4
 (8) t3     := a[t2]
 (9) if t3 < v goto (7)
(10) t4     := t4 - 4
(11) t5     := a[t4]
(12) if t5 > v goto (10)
(13) if t2 >= t4 goto (17)
(14) a[t2] := t5
(15) a[t4] := t3
(16) goto (7)
(17) t14    := a[t1]
(18) a[t2] := t14
(19) a[t1] := t3
```

- 11 instrucciones menos – 37 % más corto.
- 8 multiplicaciones menos – ¡y las que quedan se hacen *una* sola vez!
- 9 registros virtuales menos – 60 % menos registros.

UNIVERSIDAD SIMÓN BOLÍVAR

Bibliografía

- [*Aho*]
  - Sección 9.1
  - Ejercicios 9.1.1 a 9.1.4

UNIVERSIDAD SIMÓN BOLÍVAR