MO615B - Implementação de Linguagens II
e
MC900A - Tópicos Especiais em Linguagem de Programação

Prof. Sandro Rigo

www.ic.unicamp.br/~sandro

Conceitos de Otimização de Código

Introdução

- Melhorar o algoritmo é tarefa do programador
- O compilador pode ser útil para
  - Aplicar transformações que tornam o código gerado mais eficiente
  - Deixa o programador livre para escrever um código limpo

Quick Sort

```
void quicksort(m,n)
int m,n;
{
     int i,j;
     int v,x;
     if ( n <= m ) return;
     /* fragment begins here */
     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;
     /* fragment ends here */
     quicksort(m,j); quicksort(i+1,n);
}
```

Fig. 10.2. C code for quicksort.

Quick Sort

```
(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
```

Fig. 10.4. Three-address code for fragment in Fig. 10.2.

Quick Sort

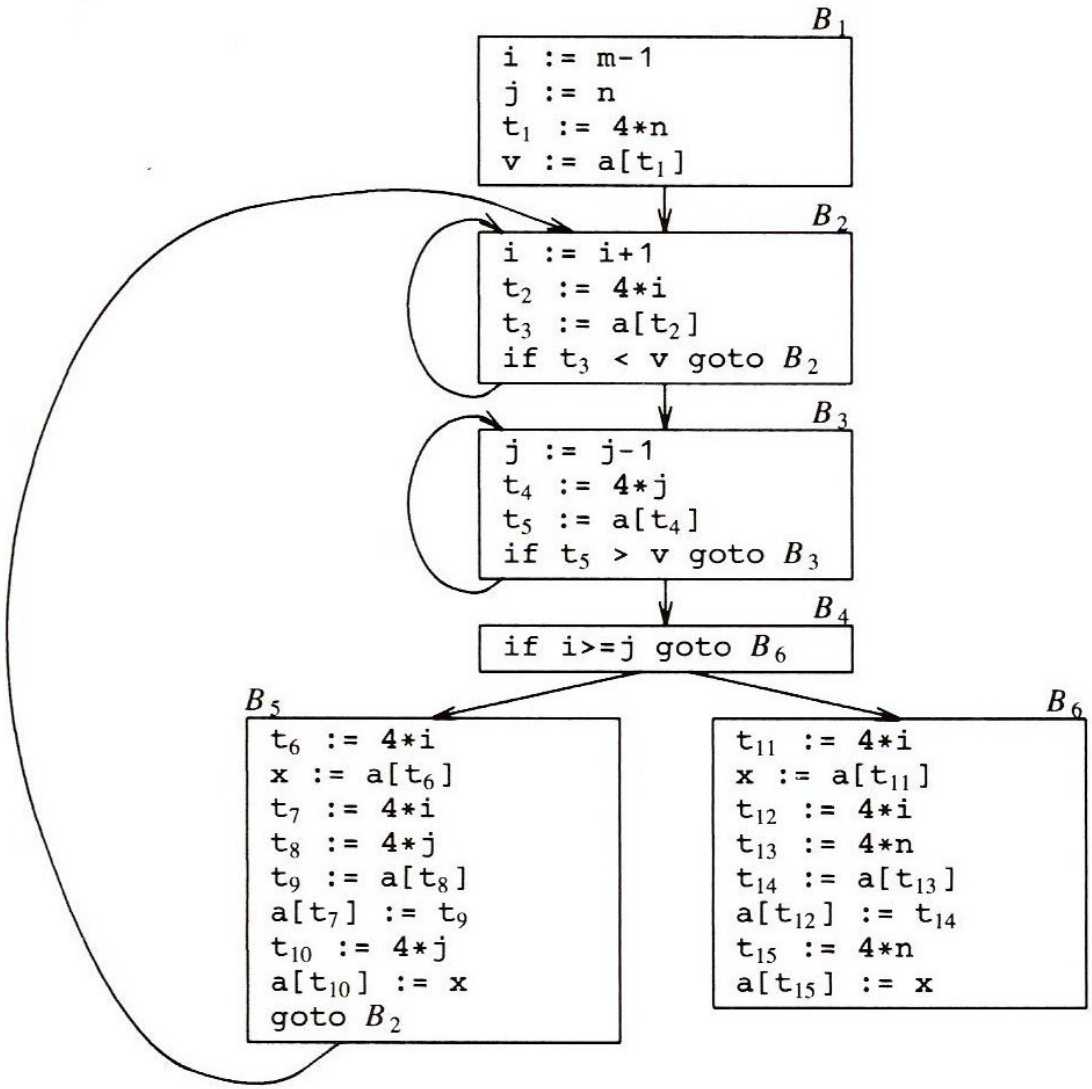


Fig. 10.5. Flow graph.

Principais Fontes de Otimização

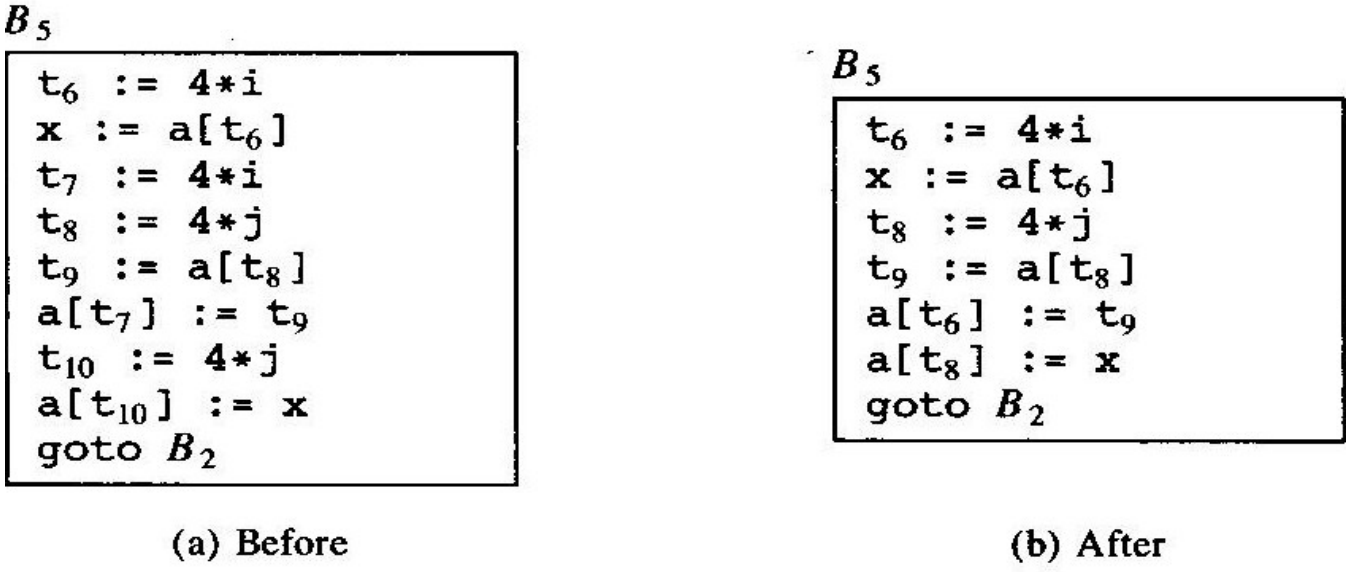
- Transformações que preservam a funcionalidade
  - Eliminação de Sub-expressões comuns (CSE)
  - Propagação de Cópias
  - Eliminação de código morto
  - Constant folding
- Transformações Locais
  - Dentro de um bloco básico
- Transformações Globais
  - Envolve mais de um bloco básico
- Livro do dragão: seção 9.1

Principais Fontes de Otimização

- Muitas podem ser tanto locais como globais
- Locais normalmente são aplicadas primeiro

Local CSE

- *E* é sub-expressão comum se
  - *E* foi previamente computada
  - Os valores usados por E não sofreram alterações



(a) Before (b) After

Fig. 10.6. Local common subexpression elimination.

Global CSE

- Podemos utilizar valores que foram definidos em outros blocos?
- Exemplo:
  - Em B2 nós computamos t2 = 4*i,
  - Poderíamos usar t2 em B5 em vez de t6...
- Como ficaria o CFG após a aplicação de "Global CSE"?

Copy Propagation

- Voltemos ao bloco B5
  - Dá para melhorá-lo ainda mais?

Dead-Code Elimination

- Código morto
  - Sentenças que computam valores que nunca são usados
- Pode ser inserido
  - Pelo programador
    - If(debug) {...}
  - Por outras transformações
    - Copy propagation
      - Bloco B5 do exemplo anterior
- Como fica o bloco B5 após aplicarmos DCE?

Otimizações em Laços

- Lugar muito importante para otimizar
- Inner loops
  - Laços mais internos
  - Tendem a ser onde os programas gastam a maior parte de seu tempo de execução
- Três técnicas básicas:
  - Code Motion
  - Induction variable elimination
  - Strenght reduction

Code Motion

- Invariantes do laço (loop-invariant)
  - Expressões que geram o mesmo resultado independente do número de iterações
- Mover a expressão “limit-2” para fora do laço
- EX:

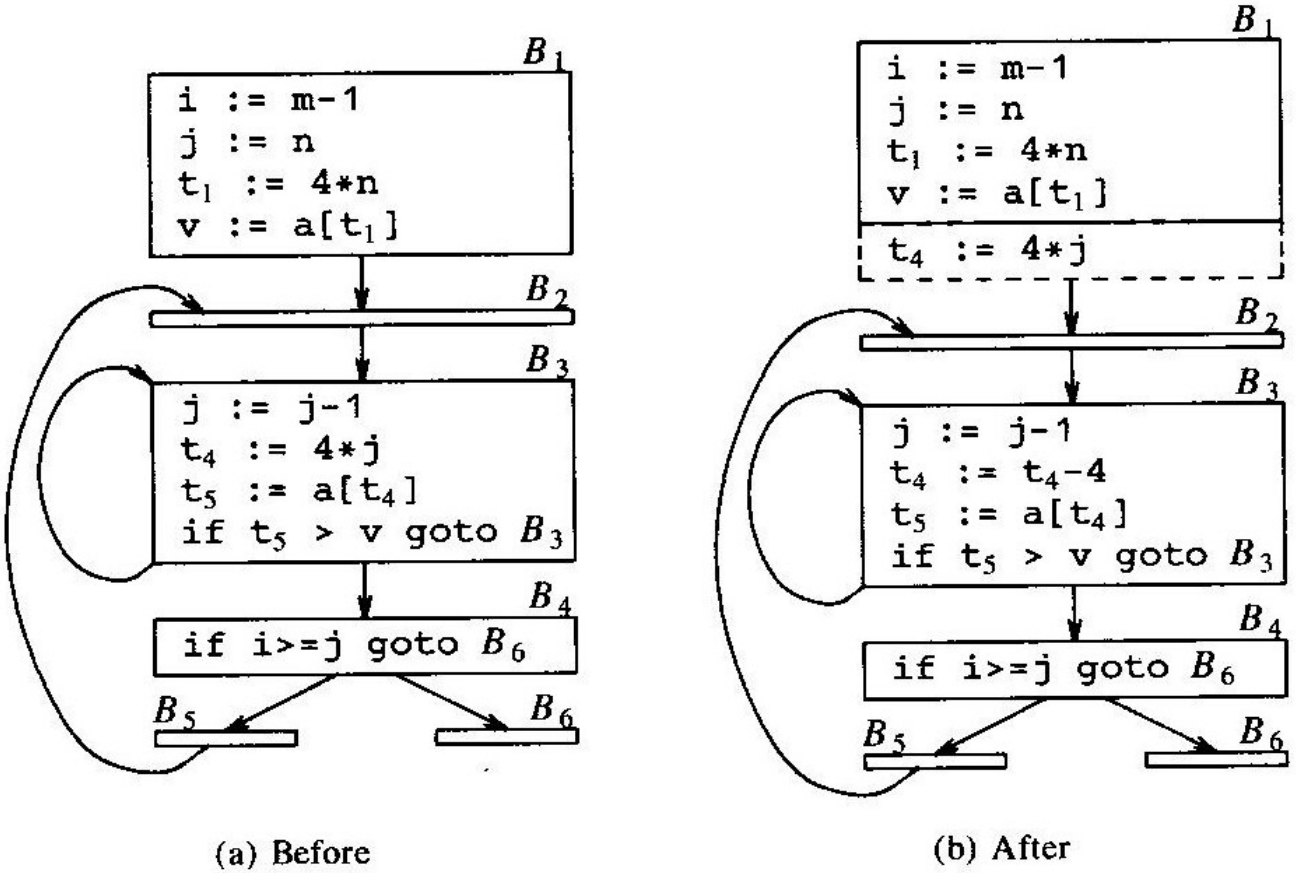
```
while (i <= limit - 2)
  { ... não altera limit ... }
```

Code Motion

- Após a transformação temos:

```
t = limit - 2
while (i <= t)
 { ... não altera limit,
   nem t ...
 }
```

Strength Reduction

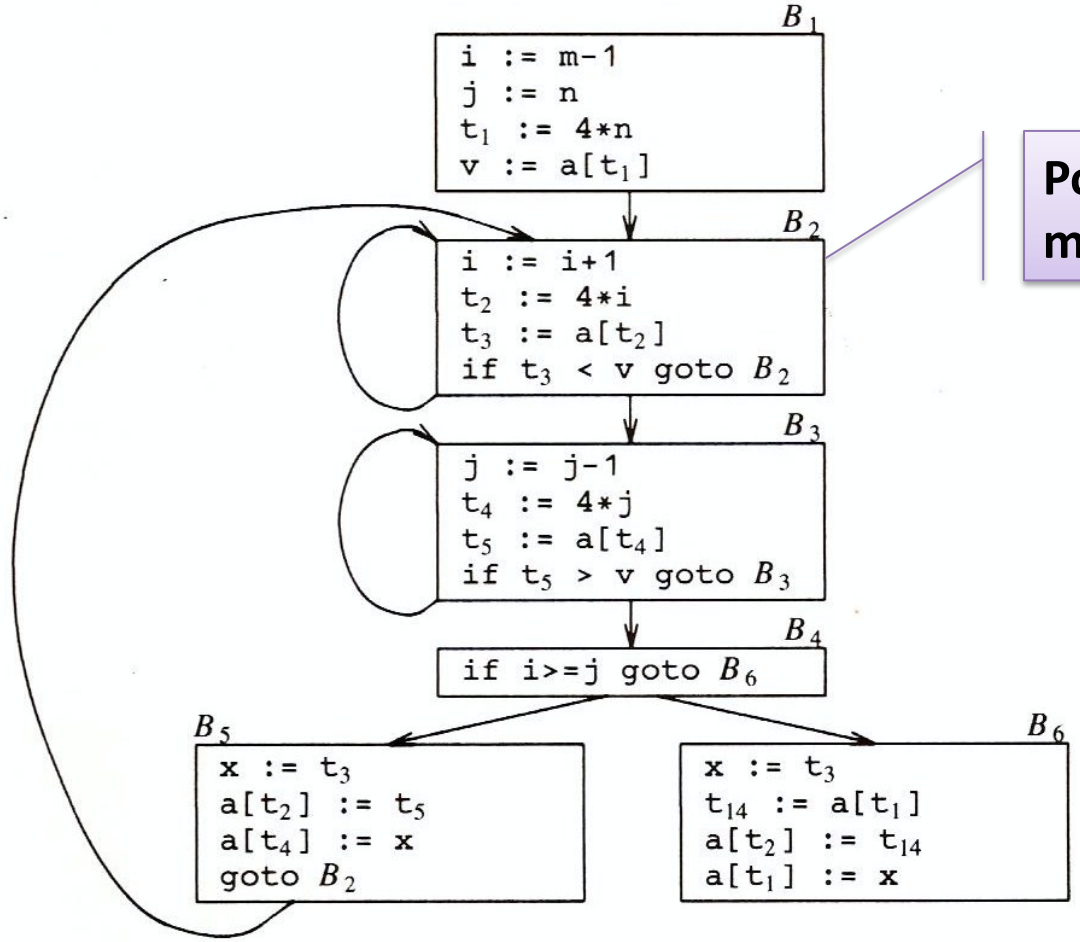


(a) Before

(b) After

Fig. 10.9. Strength reduction applied to `4*j` in block B_3.

Strength Reduction



Podemos fazer o mesmo para i e t2

Fig. 10.7. B_5 and B_6 after common subexpression elimination.

Induction Variable Elimination

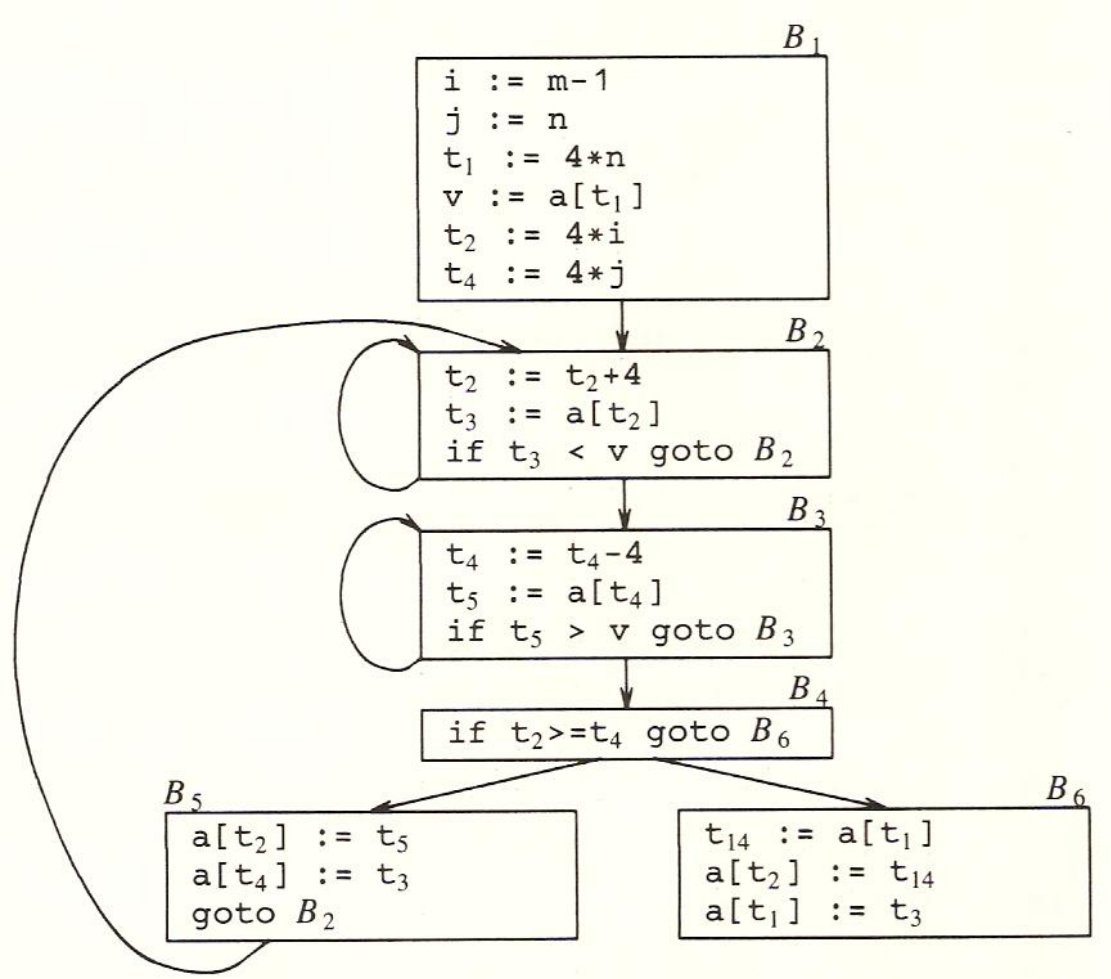


Fig. 10.10. Flow graph after induction-variable elimination.