Aula IV

Compiladores otimizadores

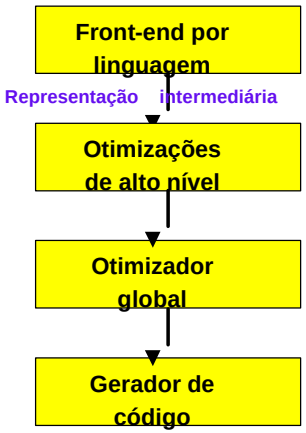
Compiladores: motivação

- **Motivação**
  - **Para entender desempenho**
  - **Tecnologia de compilação é crucial**
- **Papéis de um compilador contemporâneo**
  - **Verificador**
  - **Tradutor**
  - **Otimizador**

Objetivos de um compilador

- **Geração de código correto**
  - **Automação deve garantir correção**
- **Velocidade do código gerado**
  - **Desempenho do programa aplicativo**
- **Velocidade do compilador**
  - **Produtividade do desenvolvedor**
- **Tamanho do código gerado**
  - **Ocupação de memória**

Compilador-otimizador: estrutura



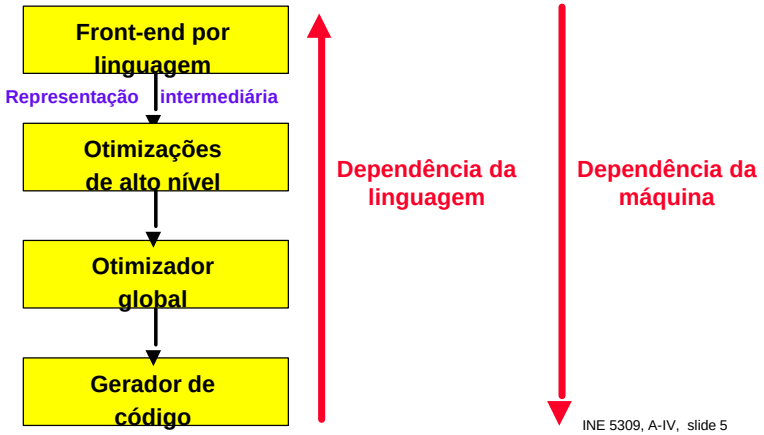
Traduzir linguagem para formato intermediário comum

Otimização de laços e inserção de in-lining

Otimizações locais, globais e alocação de registradores

Seleção de instruções e escalonamento de código

Compilador-otimizador: estrutura



Otimizações de alto nível

- **Integração de procedimentos ("inlining")**
  - **Exemplo:**
    Definição: `inline int metodo(a,b,c) {return a+b-c;}`
    Chamada: `z = metodo(w,x,y);`
    Resultado: `z = w+x-y;`
- **Transformação de laços**
  - **Exemplo: "loop unrolling"**
    » **Antes:**

```
for (i=0; i < N; i=i+1)   { a[i]=b[i]*c[i];}
```

    » **Depois:**

```
for (i=0; i < N; i=i+2) { a[i]=b[i]*c[i];
                          a[i+1]=b[i+1]*c[i+1];}
```

Otimizações de alto nível

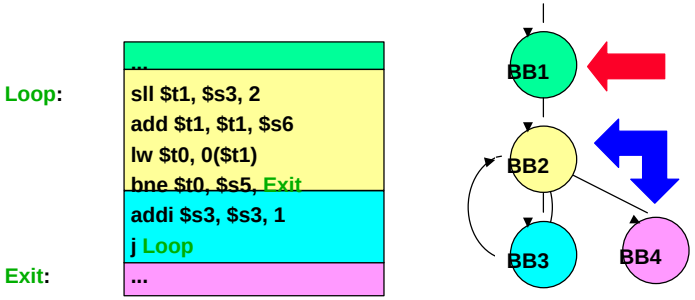
- **Integração de procedimentos (“inlining”)**
  - **Exemplo:**
    - **Definição:** `inline int metodo(a,b,c) {return a+b-c;}`
    - **Chamada:** `z = metodo(w,x,y);`
    - **Resultado:** `z = w+x-y;`
- **Transformação de laços**
  - **Exemplo: “loop unrolling”**
    - » **Antes:**

```
for (i=0; i < N; i=i+1)   { a[i]=b[i]*c[i];}
```

    - » **Depois:**

```
for (i=0; i < N/2; i=i+1) { a[i*2]=b[i*2]*c[i*2];
  a[i*2+1]=b[i*2+1]*c[i*2+1];}
```


Otimizações globais e locais

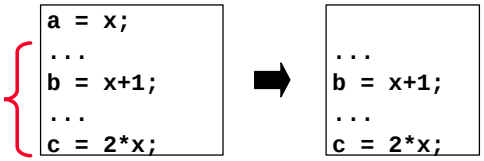


Otimização: eliminação de redundâncias ou reordenamento de instruções

Local: em BB **Global: entre BBs**

Otimizações globais e locais

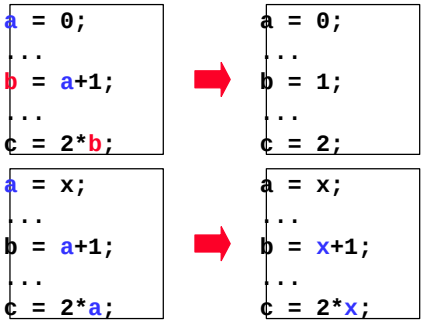
- **Eliminação de código morto**



"a" não é referenciada em instrução alguma que a sucede

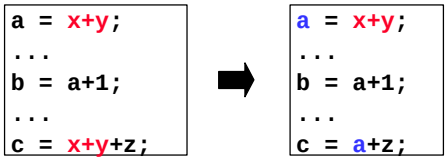
Otimizações globais e locais

- **Propagação de constantes e de cópias**



Otimizações globais e locais

- **Eliminação de expressões comuns**



Eliminação de expressões comuns

```
x[i] = x[i] + 4;
```

```
# x[i]+4
la R100,x
lw R101,i
mult R102,R101,4
add R103,R100,R102
lw R104,0(R103)
add R105,R104,4
```

Eliminação de expressões comuns

```
x[i] = x[i] + 4;
```

```
# x[i]+4
la R100,x
lw R101,i
mult R102,R101,4
add R103,R100,R102
lw R104,0(R103)
add R105,R104,4
# x[i] =
la R106,x
lw R107,i
mult R108,R107,4
add R109, R106, R107
sw R105,0(R109)
```


Eliminação de expressões comuns

```
x[i] = x[i] + 4;
```

```
# x[i]+4
la R100,x
lw R101,i
mult R102,R101,4
add R103,R100,R102
lw R104,0(R103)
add R105,R104,4
# x[i] =
la R106,x
lw R107,i
mult R108,R107,4
add R109, R106, R107
sw R105,0(R109)
```

```
# x[i]+4
la R100,x
lw R101,i
mult R102,R101,4
add R103,R100,R102
lw R104,0(R103)
add R105,R104,4
# x[i] =


sw R105,0(R103)
```

Redução de força

```
x[i] = x[i] + 4;
```

```
# x[i]+4
la R100,x
lw R101,i
mult R102,R101,4
add R103,R100,R102
lw R104,0(R103)
add R105,R104,4
# x[i] =
la R106,x
lw R107,i
mult R108,R107,4
add R109, R106, R107
sw R105,0(R109)
```

```
# x[i]+4
la R100,x
lw R101,i
mult R102,R101,4
add R103,R100,R102
lw R104,0(R103)
add R105,R104,4
# x[i] =

sw R105,0(R103)
```


Redução de força

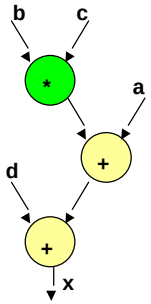
```
x[i] = x[i] + 4;
```

```
# x[i]+4
la R100,x
lw R101,i
mult R102,R101,4
add R103,R100,R102
lw R104,0(R103)
add R105,R104,4
# x[i] =
la R106,x
lw R107,i
mult R108,R107,4
add R109, R106, R107
sw R105,0(R109)
```

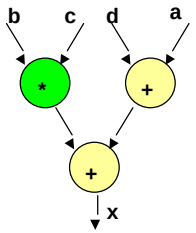
```
# x[i]+4
la R100,x
lw R101,i
sll R102,R101,2
add R103,R100,R102
lw R104,0(R103)
add R105,R104,4
# x[i] =

sw R105,0(R103)
```

Redução da altura da pilha

x = a+b*c+d; ➡ **x = (a+d)+b*c**




```
mult t0, b, c
add t1, t0, a
add x, t1, d
```




```
add t0, a, d
mult t1, b, c
add x, t0, t1
```

Podem ser executadas em paralelo

Escalonamento de código

```
lw      $t1, 0($t0)
lw      $t2, 4($t0)
add     $t3, $t1, $t2
sw      $t3, 12($t0)
lw      $t4, 8($t0)
add     $t5, $t1, $t4
sw      $t5, 16($t0)
```

Como veremos futuramente...

Em uma micro-arquitetura com pipeline, essas dependências impedem que uma instrução seja iniciada a cada ciclo.

(o que aumenta o CPI)

Escalonamento de código

```
lw    $t1, 0($t0)
lw    $t2, 4($t0)
add   $t3, $t1, $t2
sw    $t3, 12($t0)
lw    $t4, 8($t0)
add   $t5, $t1, $t4
sw    $t5, 16($t0)
```

Solução: espaçar as instruções produtoras das consumidoras, intercalando instruções independentes

Método: Reordenamento do código original, preservando as dependências de dados

Escalonamento de código

```
lw    $t1, 0($t0)
lw    $t2, 4($t0)
add   $t3, $t1, $t2
sw    $t3, 12($t0)
lw    $t4, 8($t0)
add   $t5, $t1, $t4
sw    $t5, 16($t0)
```

Escalonamento de código

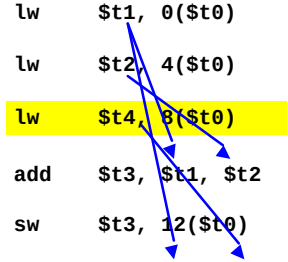
```
lw     $t1, 0($t0)
lw     $t2, 4($t0)
add    $t3, $t1, $t2
sw     $t3, 12($t0)

add    $t5, $t1, $t4
sw     $t5, 16($t0)
```


Escalonamento de código

```
lw     $t1, 0($t0)
lw     $t2, 4($t0)

add    $t3, $t1, $t2
sw     $t3, 12($t0)
add    $t5, $t1, $t4
sw     $t5, 16($t0)
```

Escalonamento de código

```
lw     $t1, 0($t0)
lw     $t2, 4($t0)
lw     $t4, 8($t0)
add    $t3, $t1, $t2
sw     $t3, 12($t0)
add    $t5, $t1, $t4
sw     $t5, 16($t0)
```



Agora uma instrução pode ser iniciada a cada ciclo

“Code motion”: invariante do laço

```
for(i=0; i<100; i=i+1)
  for(j=i; j<100; j=j+1)
       a[i,j] = 100*n + 10*(n+2) * i + j;
```

“Code motion”: invariante do laço

```
for(i=0; i<100; i=i+1)
  for(j=i; j<100; j=j+1)
       a[i,j] = 100*n + 10*(n+2) * i + j;

t1= 100*n;
t2= 10*(n+2);
for(i=0; i<100; i=i+1)
  for(j=i; j<100; j=j+1)
       a[i,j] = t1 + t2 * i + j;
```


“Code motion”: invariante do laço

```
for(i=0; i<100; i=i+1)
  for(j=i; j<100; j=j+1)
       a[i,j] = 100*n + 10*(n+2) * i + j;
```

30.000 multiplicações

```
t1= 100*n;
t2= 10*(n+2);
for(i=0; i<100; i=i+1)
  for(j=i; j<100; j=j+1)
       a[i,j] = t1 + t2 * i + j;
```

10.000 multiplicações

```
t1= 100*n;
t2= 10*(n+2);
for(i=0; i<100; i=i+1)
  t3 = t1 + t2 * i
  for(j=i; j<100; j=j+1)
       a[i,j] = t3 + j;
```

Otimização inerentemente global!

100 multiplicações

Eliminação de variável de indução

```
int a[101];
for (i=1; i£ 100; i=i+1)
  a[i] = 202 – 2*i;
```

Duas seqüências produzidas:

```
i: 1, 2, ..., 100
p: &a, &a+4, ..., &a+396
(p é endereço-base de a[])
```

Seqüências correlatas:

```
p = &a + 4*i - 4
```

Conclusão:
uma das seqüências é redundante

```
int a[101];
for (i=1; i£ 100; i=i+1)
  a[i] = 202 – 2*i;
```

```
int a[101];
t1 = 202
for (i=1; i£ 100; i=i+1)
  a[i] = t1 – 2*i;
```

Code motion do invariante

Eliminação de variável de indução

```
int a[101];
for (i=1; i£ 100; i=i+1)
  a[i] = 202 – 2*i;
```

Duas seqüências produzidas:

```
i: 1, 2, ..., 100
p: &a, &a+4, ..., &a+396
(p é endereço-base de a[])
```

Seqüências correlatas:

```
p = &a + 4*i - 4
```

Conclusão:
uma das seqüências é redundante

```
int a[101];
for (i=1; i£ 100; i=i+1)
  a[i] = 202 – 2*i;
```

```
int a[101];
t1 = 202;
for (i=1; i£ 100; i=i+1)
  a[i] = t1 – 2*i;
```

Redução de força

```
int a[101];
t1 = 202
for (i=1; i£ 100; i=i+1)
  t1 = t1 - 2;
  a[i] = t1;
```

Eliminação de variável de indução

```
int a[101];
for (i=1; i£ 100; i=i+1)
  a[i] = 202 – 2*i;
```

⟺

```
int a[101];
t1 = 202;
for(i=1; i£ 100; i=i+1)
  t1 = t1 - 2;
  a[i] = t1;
```

Variável de indução eliminada

```
int a[101];
t1 = 202;
for(p=&a; p£(&a+396);p=p+4)
  t1 = t1 - 2;
  *p = t1;
```

Duas seqüências produzidas:

~~i: 1, 2, ..., 100~~

p: &a, &a+4, ..., &a+396

(p é endereço-base de a[])

Seqüências correlatas:

p = &a + 4*i - 4

Conclusão:

uma das seqüências é redundante

Eliminação de variável de indução

```
int a[101];
for (i=1; i£ 100; i=i+1)
  a[i] = 202 – 2*i;
```

⟺

```
int a[101];
t1 = 202;
for(p=&a; p£(&a+396);p=p+4)
  t1 = t1 - 2;
  *p = t1;
```

Eliminação de variável de indução

```
int a[101];
for (i=1; i£ 100; i=i+1)
   a[i] = 202 – 2*i;
```

⟺

```
int a[101];
t1 = 202;
for( p=&a; p£(&a+396) ;p=p+4)
   t1 = t1 - 2;
   *p = t1;
```

```
        move i, 1
L1:     bgt i, 100, L2
        sll $t1, $t1, 1
        sub $t1, 202, $t1
        la $t3, a
        sll $t5, i, 2
        add $t6, $t3, $t5
        sw $t1, 0($t6)
        add i, i, 1
        j L1
L2:...
```

```
        move $t1, 202
        la $t3, a
        sub $t4,$t3, 4
        move $t5, 4
        move $t6, $t4
        add $t7, $t3, 396
L1:     bgt $t6, $t7, L2
        sub $t1, $t1, 2
        add $t6, $t4, $t5
        sw $t1, 0($t6)
        add $t5, $t5, 4
        j L1
L2:
```


Eliminação de variável de indução

```
int a[101];
for (i=1; i£ 100; i=i+1)
   a[i] = 202 – 2*i;
```

⟺

```
int a[101];
t1 = 202;
for( p=&a; p£(&a+396) ;p=p+4)
   t1 = t1 - 2;
   *p = t1;
```

```
        move i, 1
L1:     bgt i, 100, L2
        sll $t1, $t1, 1
        sub $t1, 202, $t1
        la $t3, a
        sll $t5, i, 2
        add $t6, $t3, $t5
        sw $t1, 0($t6)
        add i, i, 1
        j L1
L2:...
```

```
        move $t1, 202
        la $t3, a
        sub $t4,$t3, 4
        move $t5, 4
        move $t6, $t4
        add $t7, $t3, 396
L1:     bgt $t6, $t7, L2
        sub $t1, $t1, 2
        add $t6, $t4, $t5
        sw $t1, 0($t6)
        add $t5, $t5, 4
        j L1
L2:
```

Desempenho: impacto das otimizações

- "Procedure inlining" ($\bar{}$I, $\bar{}$CPI)
  - Mas tamanho de código aumenta
- "Loop unrolling" ($\bar{}$I ,$\bar{}$CPI)
  - Mas tamanho de código aumenta
- Eliminação de redundâncias ($\bar{}$I)
  - Eliminação de código morto
  - Eliminação de expressões comuns
  - Propagação de constantes e cópias
  - Eliminação de variável de indução

Diminuem tamanho de código

Desempenho: impacto das otimizações

- Redução de força ($\bar{}$CPI)
- Redução da altura da pilha ($\bar{}$CPI)
- Escalonamento de código ($\bar{}$CPI)
  - Aproveita melhor paralelismo entre instruções
- "Code motion"
  - Invariante do laço ($\bar{}$I)
  - Através de BBs não pertencentes a laços: ($\bar{}$CPI)
    - » Aumenta o paralelismo entre instruções no BB

Não aumentam tamanho de código

Pode aumentar tamanho de código

Uso prático: gcc

Tipo	Nome	característica	nível
Alto-nível	Procedure integration	Feita no código fonte, independente de ISA	**O3**
Local	Common subexpression elimination	Feita em formato intermediário, dentro de BBs	**O1**
	Constant propagation		
	Stack height reduction		
Global	Common subexpression elimination	Feita em formato intermediário, entre BBs	**O2**
	Copy propagation		
	Loop-invariant code motion		
	Induction variable elimination		
Dependente de processador	Strenght reduction	Depende do ISA	**O1**
	Pipeline scheduling	Depende da micro-arquitetura	