Compiladores

Aula 25: Otimizações Globais

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br

CEFET-MG - Campus Timoteo
Dep. Engenharia de Computação

Junho de 2026

Sumário

1. Objetivos
2. Introdução
3. Propagação
4. Expressões Comuns e Código Morto
5. Otimizações de Laços
6. Estudo de Caso
7. Referencias

Objetivos da Aula

- Compreender a diferença entre otimizações locais e globais.

Objetivos da Aula

- Compreender a diferença entre otimizações locais e globais.
- Entender como os resultados da Análise de Fluxo de Dados (Aula 24) embasam as otimizações.

Objetivos da Aula

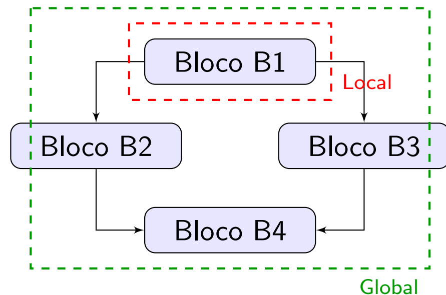
- Compreender a diferença entre otimizações locais e globais.
- Entender como os resultados da Análise de Fluxo de Dados (Aula 24) embasam as otimizações.
- Estudar as principais otimizações globais independentes da máquina:
  - Propagação de Constantes e Cópias.
  - Eliminação Global de Subexpressões Comuns (CSE).
  - Eliminação de Código Morto.

Objetivos da Aula

- Compreender a diferença entre otimizações locais e globais.
- Entender como os resultados da Análise de Fluxo de Dados (Aula 24) embasam as otimizações.
- Estudar as principais otimizações globais independentes da máquina:
  - Propagação de Constantes e Cópias.
  - Eliminação Global de Subexpressões Comuns (CSE).
  - Eliminação de Código Morto.
- Explorar transformações específicas para laços (Loop Optimizations):
  - Movimentação de Código Invariante (*Loop-Invariant Code Motion*).
  - Redução de Força (*Strength Reduction*) e Variáveis de Indução.

Otimizações: De Local a Global

- **Local:** Otimizações aplicadas isoladamente dentro de um único Bloco Básico. Mais seguras, porém com escopo limitado.
- **Global (Intraprocedural):** Otimizações aplicadas ao longo de toda a função/procedimento, considerando o Grafo de Fluxo de Controle (CFG).
- **Dependência de Dados:** Para alterar código além das fronteiras de um bloco básico com segurança, dependemos estritamente da **Análise de Fluxo de Dados** (Reaching Definitions, Liveness, etc.).



Constant Folding e Propagação de Constantes

- **Constant Folding (Dobramento):** Avaliação de expressões constantes em tempo de compilação.
  - Ex: `x = 2 + 3` $\Longrightarrow$ `x = 5`.
- **Propagação de Constantes:** Se sabemos via *Reaching Definitions* que todas as definições de y que alcançam $x = y + 1$ atribuem o valor 5 a y, podemos substituir y por 5.

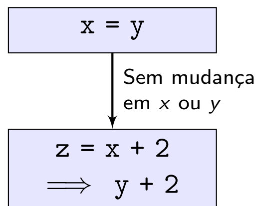
Benefícios

Substituir variáveis por constantes habilita ainda mais *Constant Folding* e simplificação de código!

Propagação de Cópias (Copy Propagation)

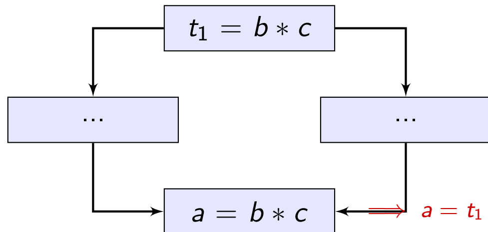
- Uma atribuição `x = y` é uma instrução de cópia.
- Se a cópia alcança um ponto p (ex: `z = x + 2`) e garantimos via fluxo de dados que nem x nem y foram redefinidos no caminho, substituímos x por y: `z = y + 2`.

Por que fazer isso?
Ao substituir usos de x por y, muitas vezes x se torna inútil e pode ser eliminada (reduz pressão nos registradores).



Global Common Subexpression Elimination (CSE)

- Baseia-se na análise de **Available Expressions**.
- Se uma expressão `b * c` está disponível antes da instrução `a = b * c`, não precisamos recalculá-la.



*Nem b nem c foram alterados nos caminhos (b*c disponível).*

Eliminação de Código Morto Global

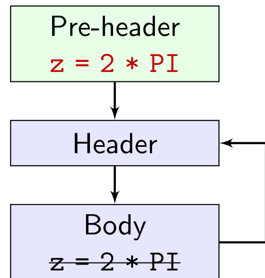
- **Código Inatingível (Unreachable Code):** Blocos no CFG que não podem ser alcançados a partir do bloco de entrada. Otimização simples por percurso de grafos (ex: BFS/DFS).
- **Instruções Inúteis (Dead Code):** Baseia-se na **Liveness Analysis**.
  - Se uma variável x sofre atribuição `x = expr`, mas x é classificada como **morta** (*dead*) nesse ponto (ninguém vai lê-la futuramente), a instrução inteira pode ser deletada!
  - Cuidado: `expr` não pode ter efeitos colaterais (ex: chamadas de função, escrita em memória/I-O).

A Importância dos Laços (Loops)

- **Regra 90/10:** Programas passam 90% do seu tempo executando 10% do código (geralmente laços).
- Melhorar o código de um laço traz enormes benefícios de desempenho global.
- Mas primeiro, como o compilador reconhece um laço no CFG?
  - Através do conceito de **Dominadores** e **Back-edges** (Arestas de retorno). Um back-edge $n \rightarrow d$ ocorre quando d domina n (todo caminho da entrada até n tem que passar por d). O nó d é o *header* do loop.

Loop-Invariant Code Motion (Movimentação de Código)

- Se uma computação `z = x op y` produz sempre o mesmo resultado em todas as iterações do laço, ela é **invariante**.
- **Estratégia:** Mover a instrução para o *pre-header* (bloco imediatamente antes do laço).
- Economiza cálculos repetidos N vezes.



Redução de Força (Strength Reduction)

- Substitui operações computacionalmente "caras" por operações "baratas" equivalentes.
- Exemplo clássico em laços: transformar multiplicações envolvendo variáveis de indução em adições.
  - Variável de Indução Básica: `i = i + 1`
  - Operação Cara: `addr = base + i * 4`
  - **Otimização:** Criar nova variável `t_addr = base` fora do laço. Dentro do laço, usar `t_addr = t_addr + 4`.
- A Adição é significativamente mais rápida que a Multiplicação no hardware.

Loop Unrolling (Desenrolamento de Laço)

- Reduz o overhead de controle do laço (testes de condição e saltos) executando múltiplas cópias do corpo do laço em uma única iteração.

Exemplo Prático

Antes:

```
for(i=0; i<100; i++) { a[i] = 0; }
```

Depois (Fator de desenrolamento 4):

```
for(i=0; i<100; i+=4) {
  a[i] = 0;
  a[i+1] = 0;
  a[i+2] = 0;
  a[i+3] = 0;
}
```

Otimização em Cascata: Passo a Passo

Compiladores executam várias passagens de otimização que alimentam umas às outras.

Código Intermediário Inicial:

```
1:  x = 10
2:  y = x + 5
3:  z = y * 2
4:  if (x > 0) goto L1
5:  print(z)
6:  L1:  w = z + 1
```

A Magia dos Compiladores!

Neste exemplo, veremos como o código à esquerda é transformado drasticamente.

A cada passagem, o compilador coleta dados de fluxo, simplifica expressões e remove código redundante. Avance para ver a transformação.

Otimização em Cascata: Passo a Passo

Compiladores executam várias passagens de otimização que alimentam umas às outras.

Código Intermediário Inicial:

```
1:  x = 10
2:  y = x + 5
3:  z = y * 2
4:  if (x > 0) goto L1
5:  print(z)
6:  L1:  w = z + 1
```

Passo 1: Propagação de Constantes

Substituímos o uso de `x` pelo valor constante 10.

```
1:  x = 10
2:  y = 10 + 5        ← Propagado
3:  z = y * 2
4:  if (10 > 0) goto L1   ← Propagado
5:  print(z)
6:  L1:  w = z + 1
```


Prof. Aléssio Miranda Júnior - alessio@cefetmg.br

Otimização em Cascata: Passo a Passo

Compiladores executam várias passagens de otimização que alimentam umas às outras.

Código Intermediário Inicial:

```
1:  x = 10
2:  y = x + 5
3:  z = y * 2
4:  if (x > 0) goto L1
5:  print(z)
6:  L1:  w = z + 1
```

Passo 2: Constant Folding (Dobramento)

Avaliamos as expressões constantes.

```
1:  x = 10
2:  y = 15                ← Avaliado
3:  z = y * 2
4:  if (true) goto L1     ← Avaliado
5:  print(z)
6:  L1:  w = z + 1
```

Otimização em Cascata: Passo a Passo

Compiladores executam várias passagens de otimização que alimentam umas às outras.

Código Intermediário Inicial:

```
1:   x = 10
2:   y = x + 5
3:   z = y * 2
4:   if (x > 0) goto L1
5:   print(z)
6:   L1:  w = z + 1
```

Passo 3: Propagação de Constante (novamente)

Agora y é uma constante (15).

```
1:   x = 10
2:   y = 15
3:   z = 15 * 2   ← Propagado
4:   if (true) goto L1
5:   print(z)
6:   L1:  w = z + 1
```

Otimização em Cascata (Cont.)

Código após Passo 3:

```
1:  x = 10
2:  y = 15
3:  z = 30    ← Dobrado
4:  if (true) goto L1
5:  print(z)
6:  L1:  w = z + 1
```

Passo 4: Eliminação de Código Inatingível

O salto na linha 4 é incondicional (`true`). A linha 5 nunca será alcançada.

```
1:  x = 10
2:  y = 15
3:  z = 30
4:  goto L1    ← Simplificado
               ← print() removido!
6:  L1:  w = 30 + 1    ← Prop. Constante
```

Código após Passo 3:

```
1:  x = 10
2:  y = 15
3:  z = 30    ← Dobrado
4:  if (true) goto L1
5:  print(z)
6:  L1:  w = z + 1
```

Passo 5: Eliminação de Código Morto

Sem o `print`, as variáveis `x`, `y`, `z` não são mais usadas (estão mortas).

```
1:  x = 10    ← Removido
2:  y = 15    ← Removido
3:  z = 30    ← Removido
4:  goto L1
6:  L1:  w = 31    ← Dobrado
```

Otimização em Cascata (Cont.)

Código após Passo 3:

```
1:  x = 10
2:  y = 15
3:  z = 30    ← Dobrado
4:  if (true) goto L1
5:  print(z)
6:  L1:  w = z + 1
```

Resultado Final:

O compilador transformou um bloco condicional de 6 instruções em apenas 1 atribuição direta!

Código Final

```
w = 31
```

Isso demonstra a poderosa sinergia entre as múltiplas passagens de otimização!

Referencias

- Aho, et al. *Compilers: Principles, Techniques, and Tools* (Livro do Dragao). Cap. 9.
- Cooper & Torczon. *Engineering a Compiler*. Cap. 8 e 10.
- Appel, Andrew W. *Modern Compiler Implementation*.

Duvidas?

Obrigado!

Email: alessio@cefetmg.br