9 – Optimização de Código

Linguagens e Programação

Ana Madureira
Engenharia Informática
Ano Letivo: 2021/2022

Fontes:

1. Processadores de Linguagens – da concepção à implementação, Rui Gustavo Crespo. IST Press.1998.
Cap. 9 Optimização de Código

2. Compiladores Princípios,Técnicas e Ferramentas AlfredV.Aho,R.Sethi e Jeffrey D.Ullman, 2007.
Cap. 10 Optimização de código

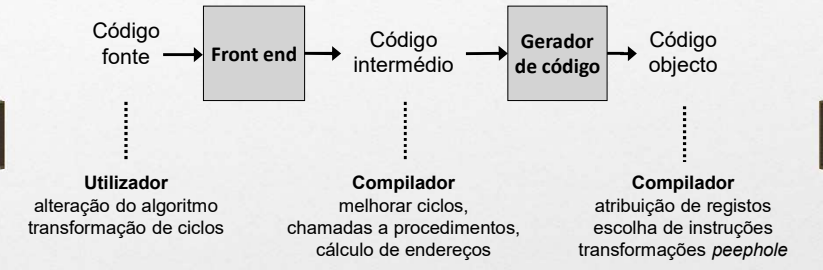
1

Optimização de Código

- Um optimizador de código pode, opcionalmente, estar presente para melhorar o código (intermédio ou objecto) tanto em termos de velocidade, de espaço, ou ambos
- A optimização automática pode ser feita tipicamente em três ocasiões:
  1. na representação intermediária, antes da geração de código
  2. durante o processo de geração de código, que já é gerado de forma optimizada
  3. após a geração de código directamente no código objecto

2

Entidade responsável pela optimização



Utilizador
alteração do algoritmo
transformação de ciclos

Compilador
melhorar ciclos,
chamadas a procedimentos,
cálculo de endereços

Compilador
atribuição de registos
escolha de instruções
transformações *peephole*

3

Propriedades do processo de optimização

- O melhor conjunto de transformações de optimização é aquele que permite alcançar o melhor benefício através do menor esforço
- Propriedades presentes:
  - A optimização tem por objectivo melhorar algum aspecto do código gerado (não encontrar o óptimo)
  - Os aspectos geralmente considerados são o espaço ocupado, o tempo de execução, etc
  - A transformação deve preservar o significado do programa – a optimização não deve alterar a saída do programa
  - "A transformação deve valer a pena"
- Geralmente consideram-se três níveis de optimização:
  - **Optimização local** - aplica-se apenas a um bloco básico
  - **Optimização global** - aplica-se a todos os blocos básicos de um procedimento ou função
  - **Optimização inter-procedimental** - aplica-se a todos os procedimentos de uma unidade de compilação
- A maior parte dos compiladores efectua a 1ª optimização; os compiladores comerciais efectuam também a 2ª; e apenas alguns fazem algum trabalho quanto à 3ª

4

Optimização durante a geração de código objecto

- Dependente da máquina alvo
- Requer uma análise do fluxo de dados (como por exemplo a análise do tempo de vida das variáveis)
- Um dos problemas a ser resolvido na geração de código é o da selecção da *instrução*: escolher uma instrução adequada entre as diversas instruções possíveis (por exemplo, com diversos modos de endereçamento)
- Outro problema é o da *alocação dos registos*, que consiste na determinação das posições onde serão armazenados os dados durante a execução das instruções (os registos). Normalmente considera-se uma hierarquia de posições possíveis:
  - os registos propriamente ditos (na CPU)
  - o topo da pilha
  - a memória em geral.

5

Optimização aplicada à representação intermédia

- Independente da máquina
- Oportunidades de optimização (optimizações locais):
  - eliminação de sub-expressões comuns
  - eliminação de código morto
  - optimização de ciclos
  - propagação de cópia
  - transformações algébricas
  - redução de força

6

Eliminação de sub-expressões comuns

Admitamos que a mesma expressão ocorre mais de uma vez num trecho de programa. Se as variáveis que ocorrem na expressão não têm os seus valores alterados entre as duas ocorrências, é possível calcular seu valor apenas uma vez. Exemplo:

```
...
x=a+b;
...
y=a+b;
...
```

Se os valores de a e de b não são alterados, é possível guardar o valor da expressão a+b numa variável temporária (t1)

```
...
t1=a+b;
x=t1;
...
y=t1;
...
```

ou, se a variável x ainda está disponível com o mesmo valor da segunda vez que a expressão é calculada, é possível dispensar o uso da variável t1

```
...
x=a+b;
...
y=x;
...
```

7

Eliminação de código morto

Código morto (*dead code*) – código que não pode ser alcançado durante a execução de um programa ⍰ remoção do código

Algumas situações:

1. Código após uma instrução de encerramento de um programa ou de uma função. Por exemplo após uma instrução *return;*

```
int f(int x) {
return x++; /* o incremento não será executado */
}
```

2. Código após um teste com uma condição impossível de ser satisfeita. Por exemplo:

```
#define DEBUG 0
...
if(DEBUG) {
... /* este código não será executado */
}
```

8

Optimização de ciclos

- Há várias optimizações que se aplicam a ciclos, a mais comum das quais é a **transferência de cálculos invariantes do ciclo** para fora dele (vamos supor que a instrução a ser movida para antes do ciclo é **x=e;**). Para que isto possa ser feito, é necessário verificar:
  - A expressão **e** é composta apenas de constantes, ou de variáveis cujos valores não são alterados dentro do ciclo
  - Nenhum uso de x, dentro ou fora do ciclo deve ter acesso a um valor de x diferente do valor a que tinha acesso antes do original

Em particular, o valor de **x** após a saída do ciclo deve ser o mesmo do programa original.
Mesmo assim, como vimos anteriormente, é possível piorar alguns programas, quando o ciclo é executado zero vezes: a instrução dentro do ciclo não seria executada, mas fora do ciclo será sempre executada uma vez.

9

Optimização de ciclos

Um exemplo de aplicação desta optimização seria (vamos supor N>0, para simplificar):

```
for (i=0, i<N, i++)
{
k=2*N;
f(k*i);
}
```

que se transformaria em

```
k=2*N;
for (i=0, i<N, i++)
f(k*i);
```

10

Propagação de cópia / Renomeação de variáveis

Durante a geração do código intermédio podem ser introduzidas variáveis temporárias que podem não ser estritamente necessárias. Normalmente esta eliminação é feita dando outros nomes às variáveis (temporárias ou não) que vão guardar os valores temporários. Por exemplo, se tivermos no código fonte **x=a+b;**, o código intermédio será

```
t1=a+b;
x=t1;
```

e a variável t1 pode ser eliminada.

11

Propagação de cópia / Renomeação de variáveis

Na avaliação de sub-expressões comuns, podemos ficar com várias variáveis desnecessárias, associadas às várias cópias da sub-expressão. Por exemplo, se tivermos no código fonte

```
x=a+b;
y=(a+b)*c;
z=d+(a+b);
```

teremos no código intermédio

```
t1=a+b;
x=t1;
t2=a+b;
t3=t2*c;
y=t3;
t4=a+b;
t5=d+t4;
z=t5;
```

e, eliminando as duas últimas cópias de **a+b**,

```
t1=a+b;
x=t1;
t2=t1;
t3=t2*c;
y=t3;
t4=t1;
t5=d+t4;
z=t5;
```

de maneira que (neste exemplo) todas as variáveis temporárias podem ser eliminadas:

```
x=a+b;
y=x*c;
z=d+x;
```

12

Propagação de cópia / Renomeação de variáveis

Por vezes, a optimização pode ser alcançada re-usando variáveis temporárias. Isto é feito mudando os nomes de algumas ocorrências dessas variáveis. Por exemplo, se tivermos

```
x=(a+b)*(c+d);
y=(e*f)+(g*h);
```

o código intermediário seria

```
t1=a+b;
t2=c+d;
t3=t1*t2;
x=t3;
t4=e*f;
t5=g*h;
t6=t4+t5;
y=t6;
```

e poderíamos *renomear* **t4** como **t1**, **t5** como **t2** e eliminar **t3** e **t6**:

```
t1=a+b;
t2=c+d;
x=t1*t2;
t1=e*f;
t2=g*h;
y=t1+t2;
```

Uma solução menos óbvia seria usar x e y como temporárias:

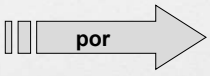
```
x=a+b;
t2=c+d;
x=x*t2;
y=e*f;
t2=g*h;
y=y+t2;
```

13

Transformações algébricas

Podemos aplicar algumas transformações baseadas em propriedades algébricas, como comutatividade, associatividade, identidade, etc. Por exemplo, como a soma é comutativa, podemos transformar **x=a+b*c;** em **x=b*c+a;** o que corresponde a trocar código como

```
Load b
Mult c
Store t1
Load a
Add t1
Store x
```




```
Load b
Mult c
Add a
Store x
```

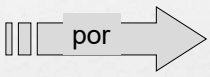
que dispensa a variável temporária t1, e as instruções que a manipulam.

14

Transformações algébricas

Transformações semelhantes podem ser baseadas na associatividade, permitindo trocar **x=(a+b)+(c+d);** por **x=(((a+b)+c)+d);** o que corresponde a trocar dispensando o uso das variáveis temporárias t1 e t2.

```
Load a
Add b
Store t1
Load c
Add d
Store t2
Load t1
Add t2
Store x
```




```
Load a
Add b
Add c
Add d
Store x
```

Em aplicações de cálculo numérico podem surgir problemas de convergência devido a erros de arredondamento quando se aplica este tipo de transformações - opções de compilação.

15

Redução de força

Há vários exemplos em que operações mais ***caras*** podem ser substituídas por operações mais ***baratas***. Por exemplo, para calcular o comprimento da concatenação de duas cadeias, podemos somar os comprimentos das duas. Em vez de

```
strlen(strcat(s1, s2))
```

usamos

```
strlen(s1) + strlen(s2)
```

Outro caso em que a redução de força é possível é no cálculo do quadrado de um número.

Se usarmos **pow(x, 2)**, em C/C++, o código gerado deverá calcular $e^{2.0 * \ln x}$, utilizando séries para calcular (aproximadamente) o logaritmo natural e a exponencial. Em vez disso, podemos usar **x*x**, com apenas uma multiplicação.

16

Optimização *Peephole*

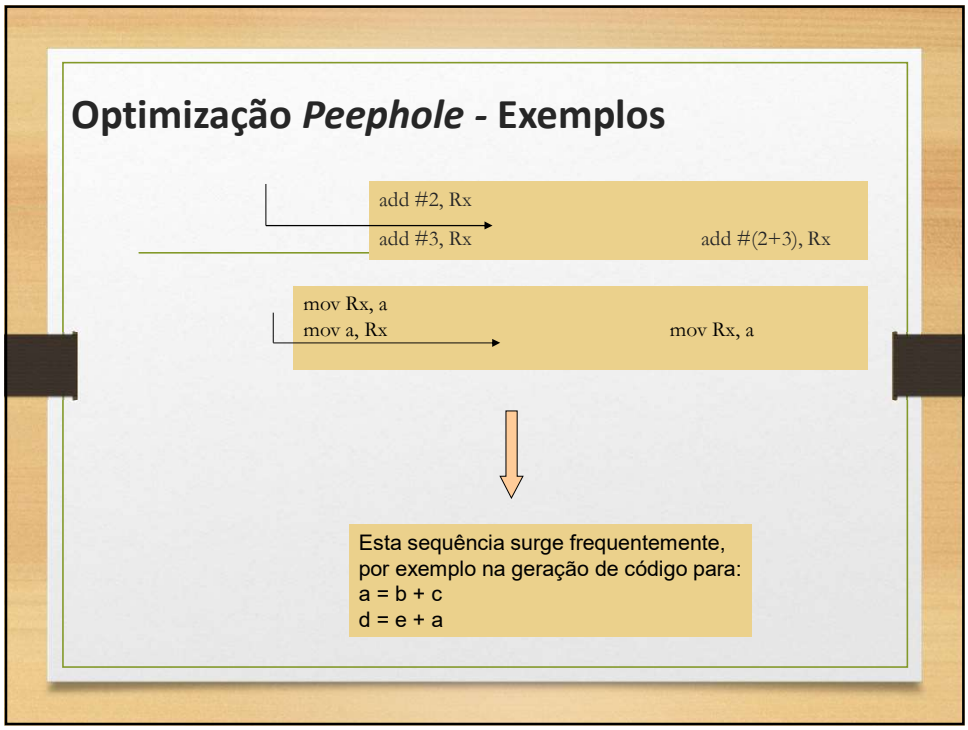
- A geração de código objecto a partir de cada instrução individual do código fonte pode originar programas que possuam redundância e construções não optimizadas
- A técnica de optimização ***Peephole*** é um método que consiste na análise de uma pequena sequência de instruções contíguas do programa (chamada *Peephole*), incluindo cerca de 3 ou 4 instruções, e sua substituição por outra sequência mais eficiente, sempre que possível
- O ***Peephole*** é uma pequena janela que vai sendo deslocada ao longo do código
- Se esse conjunto de instruções coincide com sequências pré-definidas para as quais haja um equivalente absoluto mais curto ou mais rápido faz-se a substituição
- Esta técnica é **usada frequentemente sobre o código objecto**, podendo também ser aplicada sobre o código intermédio
- Geralmente são requeridas várias passagens – a análise deve ser repetido até não haver mais substituições

17

Optimização *Peephole*

- Algumas transformações que são características deste método:
  - eliminação de instruções redundantes (eliminação de operações redundantes de load e store, eliminação de código morto, ...)
  - optimizações de fluxo de controlo (optimização de ciclos, eliminação de saltos desnecessários, ...)
  - simplificações algébricas
  - utilização de idiomas da máquina – escolha de instruções entre várias alternativas cuja execução seja mais eficiente

18


Optimização *Peephole* - Exemplos
add #2, Rx
add #3, Rx
add #(2+3), Rx
mov Rx, a
mov a, Rx
mov Rx, a
Esta sequência surge frequentemente, por exemplo na geração de código para:
a = b + c
d = e + a

19