Assembly do IA-32 em ambiente Linux

Exercícios de Programação 2

Resolução

Exercício 2.1 *(Ciclo Do-While)***:**

O código gerado na compilação de ciclos pode ser complicado de analisar, devido aos diferentes tipos de optimização do código do ciclo que o compilador poderá optar, para além da dificuldade em mapear variáveis do programa a registos do CPU. Para adquirimos alguma técnica, nada como começar com um ciclo relativamente simples.

a) A análise do modo como os argumentos são recuperados no código da sub-rotina dão-nos uma boa pista de como os registos são usados.

Utilização dos Registos		
Registo	Variável	Atribuição inicial
`%esi`	`x`	`x`
`%ebx`	`n`	`n`
`%ecx`	`y`	`y`
`%al`	`"temp1"`	`(n > 0)`
`%dl`	`"temp2"`	`(y > n)`

b) O bloco *body-statement* encontra-se nas linhas 4 a 6 no código C, e nas linhas 6 a 8 no código *assembly*. O bloco *test-expr* está na linha 7 do código C. No código *assembly*, é implementado pelas instruções nas linhas 9 a 14, bem como na condição de salto na linha 15.

c) Eis o código anotado:

```
   Inicialmente x, y, e n estão, respectivamente, à distância de 8, 12, e 16 células de %ebp
1     movl    8(%ebp),%esi      Coloque x em %esi
2     movl    12(%ebp),%ebx     Coloque y em %ebx
3     movl    16(%ebp),%ecx     Coloque n em %ecx
4     .p2align 4,,7
5  .L6:                       ciclo:
6     imull   %ecx,%ebx           y *= n
7     addl    %ecx,%esi           x += n
8     decl    %ecx                n--
9     testl   %ecx,%ecx           Teste n
10    setg    %al                 Coloque o valor lógico de (n > 0) em %al
11    cmpl    %ecx,%ebx           Compare y:n
12    setl    %dl                 Coloque o valor lógico de (y < n) em %dl
13    andl    %edx,%eax           (n > 0) & (y < n)
14    testb   $1,%al              Teste o bit menos significativo
15    jne     .L6                 Se != 0, vá para ciclo
```

Note a forma um tanto ou quanto estranha de implementar a expressão de teste: aparentemente o compilador sabe que as duas condições de teste -`(n > 0)` e `(y < n)` - apenas podem tomar os valores de 0 ou 1, e daí apenas precisa de testar o bit menos significativo do resultado do AND. O compilador poderia ter sido mais "esperto" e usado antes a instrução `testb` para efectuar a operação AND.

Exercício 2.2 *(Ciclo While)***:**

Esta é outra oportunidade de praticar a "desmontagem" do código do ciclo: o compilador de C fez algumas optimizações interessantes.

a) Tal como no exercício anterior, a análise do modo como os argumentos são recuperados no código da sub-rotina dão-nos uma boa pista de como os registos são usados e inicializados

Utilização dos Registos		
Registo	Variável	Atribuição inicial
%eax	a	a
%ebx	b	b
%ecx	i	0
%edx	result	a

b) A *test-expr* aparece na linha 5 do código C e no código *assembly* na linha 10 e na condição de salto na linha 11. O *body-statement* nas linhas 6 a 8 do código C e nas linhas 7 a 9 em *assembly*. O compilador detectou que o teste inicial do ciclo *while* será sempre verdadeiro, uma vez que `i` é inicializado a 0, o que é claramente inferior a 256.

c) Eis o código anotado:

```
1    movl    8(%ebp),%eax       Coloque a em %eax
2    movl    12(%ebp),%ebx      Coloque b em %ebx
3    xorl    %ecx,%ecx          i = 0
4    movl    %eax,%edx          result = a
5    .p2align 4,,7
   a em %eax, b em %ebx, i em %ecx, result em %edx
6 .L5:                          ciclo:
7    addl    %eax,%edx            result += a
8    subl    %ebx,%eax            a -= b
9    addl    %ebx,%ecx            i += b
10   cmpl    $255,%ecx            Compare i:255
11   jle     .L5                  Se <= vá para ciclo
12   movl    %edx,%eax            Faça de result o valor de retorno
```

d) Eis o código equivalente de `goto:`

```
1 int loop_while_goto(int a, int b)
2 {
3    int i = 0;
4    int result = a;
5    loop:
6       result += a;
7       a -= b;
8       i += b;
9       if (i <= 255)
10        goto loop;
11   return result;
12 }
```

Exercício 2.3 *(Ciclo For)***:**

Uma forma de se analisar o código *assembly* é tentar inverter o processo de compilação e produzir código C que pareça "natural" a um programador de C. Por exemplo, não queremos código com instruções `goto`, uma vez que estas são raramente usadas em C; e muito provavelmente não usaríamos também aqui a instrução `do-while`. Este exercício obriga-nos a pensar no processo inverso da compilação num dado enquadramento: no modo como os ciclos `for` são traduzidos.

Pode-se ver que:

- `result` deve estar no registo `%eax`; é colocado inicialmente a 0 e é deixado em `%eax` no fim do ciclo, como valor de retorno
- `i` é mantido no registo `%edx`, uma vez que este registo é usado como base para 2 testes condicionais; as instruções nas linhas 2 e 4 colocam `n-1` em `%edx`
- os testes nas linhas 5 e 12 requerem que `i` não seja negativo
- a variável `i` é decrementada por `x` na instrução 11
- as instruções 1, 6, e 7 fazem com que `x*y` seja armazenado no registo `%ecx`, o qual é usado para incrementar o valor da variável `result`, no interior do ciclo

a) Devido a uma técnica de optimização conhecida como *code motio*n, onde um cálculo é retirado para fora dum ciclo, quando é possível determinar que o seu resultado não variará dentro do ciclo.

b) Eis o código original:

```
1 int loop(int x, int y, int n)
2 {
3    int result = 0;
4    int i;
5    for (i = n-1; i >= 0; i = i-x) {
6       result += y * x;
7    }
8    return result;
9 }
```