

Tail Call Optimization

Brunno Perez Bocardo | Flavia Moreira Goes | Diego Floriano

- **O que é?**

Sabemos que uma função recursiva chama a si mesma, e vai se repetindo até que chegue a uma certa condição de parada. Por exemplo, se queremos mandar um mesmo email para uma lista de contatos, podemos usar uma função recursiva para enviar o email para cada um dos contatos até que a lista chegue ao fim, sendo essa a condição de parada. Mas ao invés de criar uma função recursiva, esse e outros problemas poderiam ser facilmente resolvidos por um *for*. Então, a função recursiva é de fato eficiente?

Embora muitos acreditem que ela é totalmente ineficiente, essa não é uma verdade absoluta. Claro, em algumas situações, não é muito vantajoso optar por uma recursividade, como a soma dos elementos de um vetor, ou principalmente em casos que gerem bifurcações exponenciais, como a sequência de Fibonacci.

Então, em quais casos as funções recursivas são eficientes? Bem, considerando que a cada chamada, a função ocupa diferentes posições dentro da memória, e que soluções eficazes devem gastar a menor quantidade possível de recursos, funções recursivas não são eficientes. O que resolve essa falha da recursividade é justamente o Tail Call Optimization (TCO).

Em uma função recursiva, o fluxo é sempre o mesmo. Ela segue suas operações até que chama a si mesma, e vai fazendo isso a cada chamada até atingir a condição de parada. O TCO faz com que ele reutilize um único fluxo, ou seja, um novo espaço na memória não é criado ao se chamar uma nova função. Desse modo, menos recursos são gastos.

- **Como ele funciona?**

O Tail Call Optimization é uma técnica que elimina a necessidade da criação de um novo escopo de memória para armazenar os dados de outra função. Essa técnica é

mais útil em Tail Recursion, função que apresenta uma chamada recursiva em seu código como última instrução, ao final a função chama a si mesma tornando possível e eficiente ser convertida para uma função iterativa no qual o escopo de memória da própria função é utilizado para todas as seguintes chamadas.

Na imagem abaixo é apresentada uma função recursiva que calcula um número fatorial, se o valor estabelecido para ser calculado fosse exorbitante uma função como essa causaria um estouro de pilha (quando o programa utiliza mais memória da pilha do que está disponível).

```
int fatorial(int n){
    if(n < 1) {
        return 1; //caso base
    }

    return n * fatorial(n-1);
}

int main(int argc, char *argv[]) {
    int resultado = fatorial(5);
    printf("%d\n", resultado);
    return 0;
}
```

função sem Tail Recursion

Aqui (na imagem abaixo) o TCO entra em cena, com o qual o compilador pode evitar a criação de um novo quadro de pilha, desde que a chamada feita seja a última operação na função em que é chamado. Essa técnica ajuda a evitar erros de estouro de pilha para chamadas recursivas profundas e aumenta a eficiência do programa.

```

int fatorial(int store, int n){
    if(n < 1) {
        return store; //caso base
    }

    return fatorial(n * store, n-1);
}

int main(int argc, char *argv[]) {
    int resultado = fatorial(1,5);
    printf("%d\n", resultado);
    return 0;
}

```

função com Tail Recursion

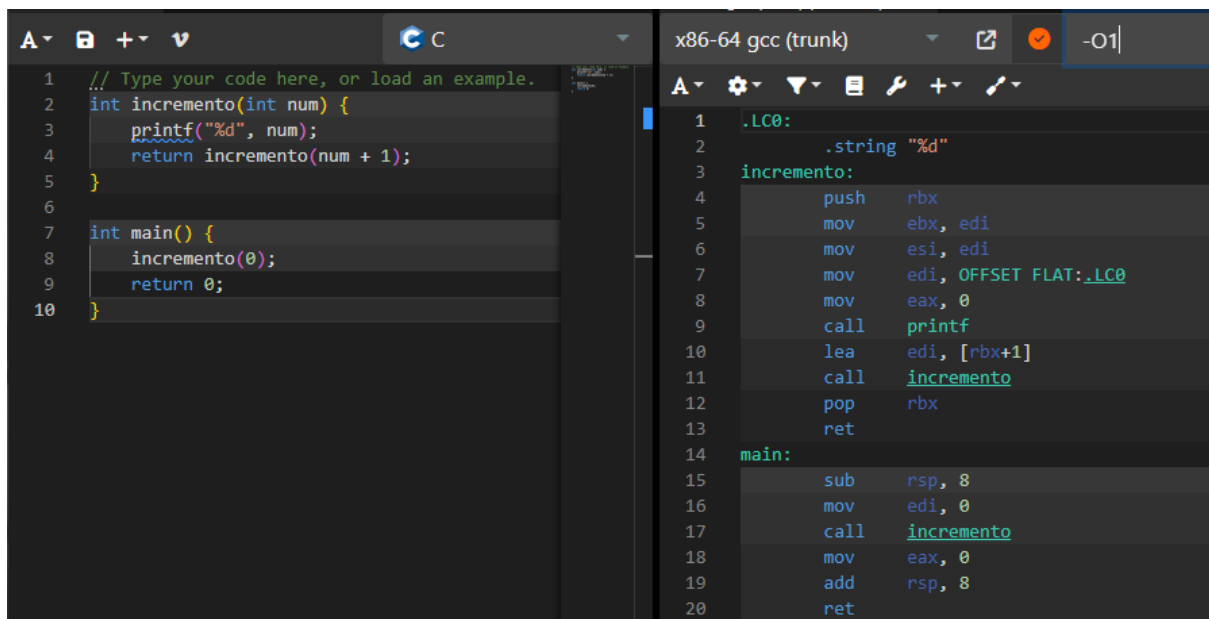
O quadro existente pode ser sobrescrito diretamente pelo compilador: o valor antigo de n é multiplicado pelo valor antigo de store e o resultado é gravado de volta na variável store. O compilador agora pode decrementar o n substituindo o valor anterior e pulando para o início da função fatorial. É garantido que, em vez da operação de multiplicação, a chamada de função seja a última operação no código acima que precisa ser executada.

Para o compilador otimizar o seu código no Dev C++, basta clicar em Project no campo superior, clicar em Project Options. Abrirá uma janela, e nessa janela no canto superior, clique em Compiler, Code Generation e em Optimization Level coloque na opção High. Em seguida clique em OK para concluir. Para ter certeza que o código será reconstruído clique em Execute no canto superior e depois em Rebuild All.

- Como o compilador transforma o código recursivo em não recursivo?

Quando é executada a função recursiva, ela volta para a mesma função sendo obrigada a criar um novo local da memória para alocar o próximo dado e assim sucessivamente até o ponto de parada, e isso é um problema como vimos anteriormente.

Para entendermos melhor como o compilador transforma código recursivo em não recursivo, podemos usar o site [GodBolt](https://godbolt.org/) para analisar o código assembly de uma função recursiva de causa comum e quando é usado o TCO.



The image shows a screenshot of the GodBolt compiler explorer interface. On the left, the C source code is displayed: a recursive function `incremento` that prints its argument and calls itself with `num + 1`, and a `main` function that calls `incremento(0)`. On the right, the generated assembly code for x86-64 GCC (trunk) is shown with optimization level `-O1`. The assembly includes a label `.LC0` for the string `"%d"`, the `incremento` function body, and the `main` function body. The `main` function sets up the stack, calls `incremento`, and then cleans up the stack before returning.

```
1 // Type your code here, or load an example.
2 int incremento(int num) {
3     printf("%d", num);
4     return incremento(num + 1);
5 }
6
7 int main() {
8     incremento(0);
9     return 0;
10 }

x86-64 gcc (trunk) -O1

1 .LC0:
2     .string "%d"
3 incremento:
4     push    rbx
5     mov     ebx, edi
6     mov     esi, edi
7     mov     edi, OFFSET FLAT:.LC0
8     mov     eax, 0
9     call    printf
10    lea     edi, [rbx+1]
11    call    incremento
12    pop     rbx
13    ret
14 main:
15    sub     rsp, 8
16    mov     edi, 0
17    call    incremento
18    mov     eax, 0
19    add     rsp, 8
20    ret
```

Na imagem acima, podemos ver uma função recursiva que iria exibir números sequenciais de modo infinito. Ao analisarmos o assembly na direita, vemos que a função `incremento` é chamada dentro da `main`. Já dentro da própria função, ao fim dela ela chama a própria função do início.

```
1 // Type your code here, or load an example.
2 int incremento(int num) {
3     printf("%d", num);
4     return incremento(num + 1);
5 }
6
7 int main() {
8     incremento(0);
9     return 0;
10 }
```

```
1 .LC0:
2     .string "%d"
3 incremento:
4     push    rbx
5     mov     ebx, edi
6 .L2:
7     mov     esi, ebx
8     mov     edi, OFFSET FLAT:.LC0
9     xor     eax, eax
10    add     ebx, 1
11    call    printf
12    jmp     .L2
13 main:
14    sub     rsp, 8
15    xor     edi, edi
16    call    incremento
```

Já na imagem acima, no código assembly, podemos perceber que o próprio compilador cria uma nova linha (label) denominada “.L2:”, com isso a recursão em vez de voltar para a própria função, ela lê a linha de “.L2:” utilizando o mesmo espaço de memória do dado anterior, porém o modificando. Ou seja, o compilador transformou a função recursiva em não-recursiva. E isso garante que outros espaços de memória não sejam utilizados aumentando o desempenho do programa.

REFERÊNCIAS:

<https://www.youtube.com/watch?v=GZJ8A3D-ESQ/>

<https://www.geeksforgeeks.org/tail-call-optimisation-in-c/>

<https://medium.com/trainingcenter/o-que-%C3%A9-recurs%C3%A3o-e-tail-call-optimizati%C3%A3o-tco-f1938188223c>