Rohit Awate
Posted on Jul 11, 2019

Demystifying Tail Call Optimization

#tailrecursion #tailcalloptimization #compiler #c

Originally published on [my personal blog](#).

Tail call optimization (a.k.a. tail call elimination) is a technique used by language implementers to improve the recursive performance of your programs. It is a clever little trick that eliminates the memory overhead of recursion. In this post, we'll talk about how recursion is implemented under the hood, what tail recursion is and how it provides a chance for some serious optimization.

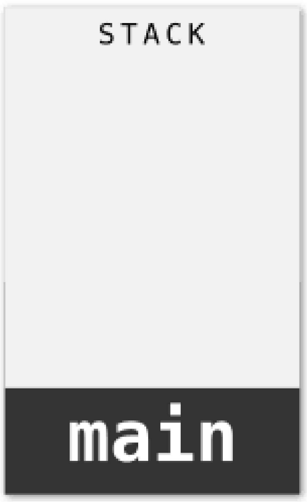
Recursion 101

If you're familiar with function call stacks and recursion, feel free to skip this section.

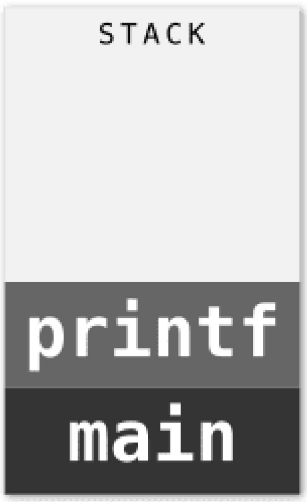
Most languages use a stack to keep track of function calls. Let's take a very simple example: a "hello, world" program in C.

```c
#include <stdio.h>

int main()
{
    printf("hello, world!\n");
    return 0;
}
```

Every function call in your program gets its own frame pushed onto the stack. This frame contains the local data of that call. When you execute the above program, the `main` function would be the first frame on the stack, since that's where your program begins execution. The topmost frame in the stack is the one currently being executed. After it completes execution, it is popped from the stack and the bottom frame resumes execution.



In our example, `main` in turn calls `printf`, another function, thereby pushing a new frame onto the stack. This frame will contain `printf`'s local data. Once `printf` completes execution, its frame is popped and control returns to the `main` frame.



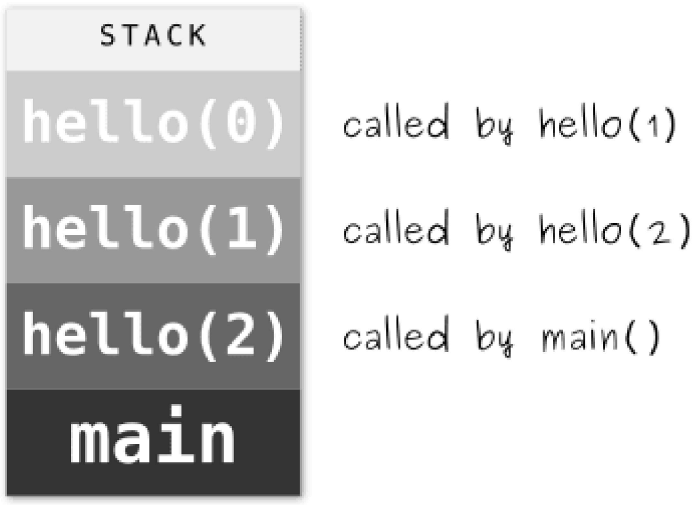
Recursive functions do the same. Every recursive call gets its own frame on the stack. Here's a *horrible example* of a recursive function which prints "hello" `n` times:

```
// hello_recursive.c
// hello_recursive.c

void hello(int n)
{
    if (n == 0) return;

    printf("hello\n");
    hello(n - 1);
}

int main()
{
    hello(2);
    return 0;
}
```

The above code gives the following output:

```
hello
hello
```

The function call stack will be something like this:



The first two calls will print out "hello" and make recursive calls with `n - 1`. Once we hit the last call with `n = 0`, we begin unwinding the stack.

Now imagine that we wish to print "hello" a million times. We'll need a million stack frames! I tried this out and my program ran out of memory and crashed. Thus, recursion requires `O(n)` space complexity, `n` being the number of recursive calls. This is bad news, since recursion is usually a natural, elegant solution for many algorithms and data structures. However, memory poses a physical limit on how tall (or deep, depending on how you look at it) your stack grows. Iterative algorithms are usually far more efficient, since they eliminate the overhead of multiple stack frames. But they can grow unwieldy and complex.

Tail Recursion

Now that we've understood what recursion is and what its limitations are, let's look at an interesting type of recursion: **tail recursion**.

Whenever the recursive call is the last statement in a function, we call it tail recursion. However, there's a catch: there cannot be any computation after the recursive call. Our `hello_recursive.c` example is tail recursive, since the recursive call is made at the very end i.e. tail of the function, with no computation performed after it.

```
    ...
    hello(n - 1);
}
```

Since this example is plain silly, let's take a look at something serious: **Fibonacci numbers**. Here's a non tail-recursive variant:

```
// Returns the nth Fibonacci number.
// 1 1 2 3 5 8 13 21 34 ...
int fib(int n)
{
    if (n <= 1) return n;
    return fib(n - 1) + fib(n - 2);
}
```

You might argue that this is tail recursive, since the recursive calls appear at the end of the function. However, the results of the calls are added *after* they return. Thus, `fib` is not tail recursive.

Here's the tail-recursive variant:

```
int fib_tail(int n, int a, int b)
{
    if (n == 0)
        return a;
    if (n == 1)
        return b;

    return fib_tail(n - 1, b, a + b);
}
```

The recursive call appears last and there are no computations following it. Cool.

You may be thinking, "*Hmm, tail recursion is interesting, but what is the point of this?*". Turns out, it is more than just a way of writing recursive functions. It opens up the possibility for some clever optimization.

Tail Call Optimization

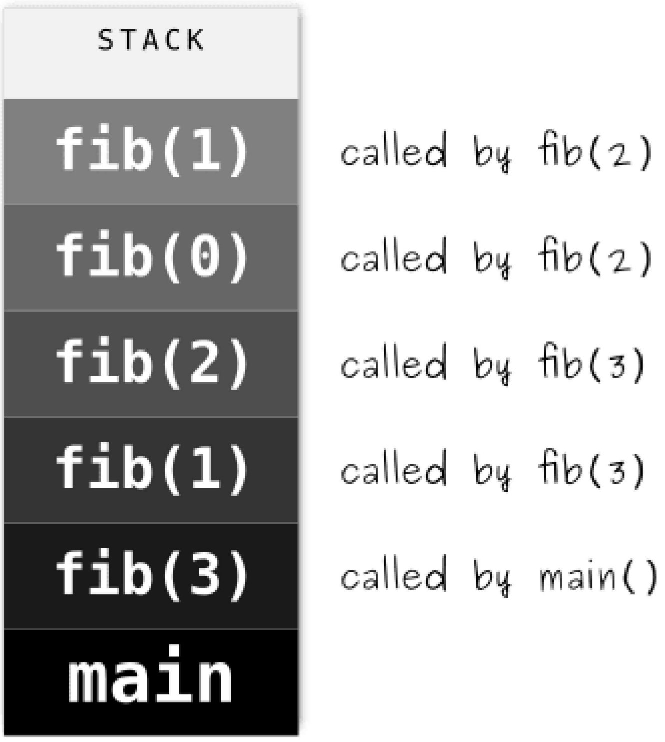
Tail call optimization reduces the space complexity of recursion from `O(n)` to `O(1)`. Our function would require constant memory for execution. It does so by eliminating the need for having a separate stack frame for every call.

If a function is tail recursive, it's either making a simple recursive call or returning the value from that call. **No computation is performed on the returned value**. Thus, there is no real need to preserve the stack frame for that call. We won't need any of the local data once the tail recursive call is made: we don't have any more statements or computations left. We can simply modify the state of the frame as per the call arguments and jump back to the first statement of the function. No need to push a new stack frame! We can do this over and over again with just one stack frame!

Let's look at our example with the non tail-recursive `fib` function. To find out the 3rd Fibonacci number, we'd do:

```
...
int third_fib = fib(3);
...
```

Assuming right-to-left precedence (i.e. the direction in which an expression is evaluated), the call stack would look something like this:



Quite large, isn't it? Imagine the size of the stack for finding out a later Fibonacci number! The problem here is that all the stack frames need to be preserved. You may use one of the local variables in the addition and hence the compiler needs to keep the frames around. If you look at the assembled output of this program, you'll see a `call` instruction for the `fib` function.

You can use the `-s` flag on GCC to output the assembly code. I've deliberately used the `-O2` flag which uses the 2nd level of optimization among GCC's 0-3 levels. O2 enables tail call optimization. If you're not familiar with assembly, use GCC's `-fverbose-asm` flag while compiling. It adds your C code as comments before its corresponding assembled output. Here's the final command, which will produce a `.s` file:

```
gcc fib.c -S -O2 -fverbose-asm
```

This is what our tail call translates to:

```
# Snippet extracted from: fib.s

# fib.c:7:  return fib(n - 1) + fib(n - 2);
movl    %ecx, %edi  # ivtmp.22,
call    fib #
```

```
 subl    $2, %ecx    #, ivtmp.22
 addl    %eax, %edx  # _4, add_acc_7
```

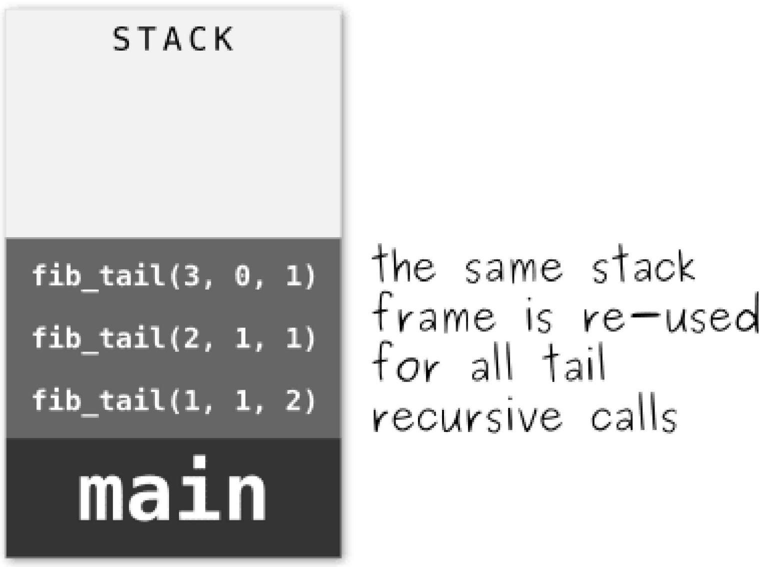
Now, I'm not going to pretend that I understand this completely, because I don't. We only care about the instructions, none of the operand details. Notice the `call fib` instruction? That's the recursive call. It pushes a new frame onto the stack. Once that completes and pops, we have our addition instruction. Thus, we conclude that even at the 2nd level of optimization, the recursive calls cannot be eliminated, thanks to the addition.

```
    ...
    return fib_tail(n - 1, b, a + b);
}
```

Let's take a look at our tail recursive Fibonacci function, `fib_tail`. Once the above recursive call is made, there's no need to keep the local data around. There's no computation following the statement and it's simply returning the value returned by the recursive call; we could do that straight from the recursive call.

This presents an opportunity to simply replace the values of the local `n`, `a` and `b` variables with the ones used in the recursive call. Instead of a `call` instruction like before, the compiler can simply redirect the flow of execution to the first instruction in the function, effectively emulating a recursive call. But, without the overhead of one! Basically, the compiler goes:

This is how the call stack would look like:


STACK
fib_tail(3, 0, 1)
fib_tail(2, 1, 1)
fib_tail(1, 1, 2)
main
the same stack frame is re-used for all tail recursive calls

You don't have to take my word for it, let's look at the assembler output for `fib_tail`. We compile the same way as before:

```
gcc fib_tail.c -S -O2 -fverbose-asm
```

For our tail recursive call, I see the following snippets of assembly:

```
# fib_tail.c:11:    return fib(n - 1, b, a + b);
    movl    %eax, %edx  # <retval>, b

# fib_tail.c:11:    return fib(n - 1, b, a + b);
    leal    (%rsi,%rdx), %eax   #, <retval>

# fib_tail.c:11:    return fib(n - 1, b, a + b);
    leal    (%rcx,%rdx), %esi   #, _5

# fib_tail.c:11:    return fib(n - 1, b, a + b);
    movl    %esi, %edx  # _5, b
```

As I said, I don't really understand assembly, but we're just checking if we've eliminated the `call fib` recursive calls. I'm not really sure how GCC is redirecting the control flow. What matters, however, is that there are no `call fib` instructions in the code. That means there are no recursive calls. The tail call has been eliminated. Feel free to dive into the assembly and verify for yourself.

Support

- We've been using **C** in this post since GCC and Clang both support tail call optimization (TCO).
- For **C++**, the case holds with Microsoft's Visual C++ also offering support.
- **Java** and **Python** do not support TCO with the intention of preserving the stack trace for debugging. Some internal Java classes also rely on the number of stack frames. Python's BDFL, Guido van Rossum, has explicitly stated that no Python implementations should support TCO.
- **Kotlin** even comes with a dedicated `tailrec` keyword which converts recursive functions to iterative ones, since the JVM (Kotlin compiles to JVM bytecode) doesn't support TCO.
- TCO is part of ECMAScript 6 i.e. the **JavaScript** specification, however, only Safari's JavaScriptCore engine supports TCO. Chrome's V8 retracted support for TCO.
- **C#** does not support TCO, however, the VM it runs within, Common Language Runtime (CLR) supports TCO.
- Functional languages such as **Haskell**, **F#**, **Scala** and **Elixir** support TCO.

It appears that support for TCO is more of an ideological choice for language implementers, rather than a technical one. It does manipulate the stack in ways the programmer would not expect and hence makes debugging harder. Refer the documentation of the specific implementation of your favorite language to check if it supports tail call optimization.

Conclusion

I hope you understood the idea and techniques behind TCO. I guess the takeaway here is to prefer iterative solutions over recursive ones *(that is almost always a good idea,* *performance-wise)*. If you absolutely need to use recursion, try to analyze how big your stack would grow with a non-tail call.

If both of these conditions don't work for you and your language implementation supports tail call optimization, go for it. Keep in mind that debugging will get harder so you might want to turn off TCO in development and only enable it for production builds which are thoroughly tested.

That's it for today, see you in the next post!

Discussion (2)

s.t. • Jan 10 '20

Little nitpick: "Assuming right-to-left precedence (i.e. the direction in which an expression is evaluated)" does *not* specify the order in which the functions are called. This is often stated (even in books) but wrong. The chosen order is implementation (aka compiler) specific and independent from the order their results are then used in the caller.

Anshul Negi • Nov 29 '20

well written
keep it up...
One question, if I add a local variable to a tail recursive function then will it allocate separate stack frame for every call?

Code of Conduct • Report abuse

Rohit Awate

Microsoft Student Partner | Developer | CS Undergrad

LOCATION
Pune, India

EDUCATION
Computer Engineering at University of Pune

JOINED
Nov 23, 2017

Trending on DEV Community 🔥

November 12th, 2021: What did you learn this week?

#weeklylearn #discuss #weeklyretro

YOU are helping Google build Self Driving cars! 🤓

#a11y #webdev #watercooler #discuss

What was your win this week?

#discuss #weeklyretro