Inlining optimisations in Go | Dave Cheney

by Dave Cheney

This is a post about how the Go compiler implements inlining and how this optimisation affects your Go code.

n.b. This article focuses on *gc*, the de facto Go compiler from golang.org. The concepts discussed apply broadly to other Go compilers like gccgo and tinygo but may differ in implementation and efficacy.

What is inlining?

Inlining is the act of combining smaller functions into their respective callers. In the early days of computing this optimisation was typically performed by hand. Nowadays inlining is one of a class of fundamental optimisations performed automatically during the compilation process.

Why is inlining important?

Inlining is important for two reasons. The first is it removes the overhead of the function call itself. The second is it permits the compiler to more effectively apply other optimisation strategies.

Function call overhead

Calling a function[1] in any language carries a cost. There are the overheads of marshalling parameters into registers or onto the stack (depending on the ABI) and reversing the process on return. Invoking a function call involves jumping the program counter from one point in the instruction stream to another which can cause a pipeline stall. Once inside the function there is usually some preamble required to prepare a new stack frame for the function to execute and a similar epilogue needed to retire the frame before returning to the caller.

In Go a function call carries additional costs to support dynamic stack growth. On entry the amount of stack space available to the goroutine is compared to the amount required for the function. If insufficient stack space is available, the preamble jumps into the runtime logic that grows the stack by copying it to a new, larger, location. Once this is done the runtime jumps back to the start of the original function, the stack check is performed again, which now passes, and the call continues. In this way goroutines can start with a small stack allocation which grows only when needed.[2]

This check is cheap–only a few instructions–and because goroutine stacks grows geometrically the check rarely fails. Thus, the branch prediction unit inside a modern processor can hide the cost of the stack check by assuming it will always be successful. In the case where the processor mis-predicts the stack check and has to discard the work done while it was executing speculatively, the cost of the pipeline stall is relatively small compared to the cost of the work needed for the runtime to grow a goroutines stack.

While the overhead of the generic and Go specific components of each function call are well optimised by modern processors using speculative execution techniques, those overheads cannot be entirely eliminated, thus each function call carries with it a performance cost over and above the time it takes to perform useful work. As a function call's overhead is fixed, smaller functions pay a larger cost relative to larger ones because they tend to do less useful work per invocation.

The solution to eliminating these overheads must therefore be to eliminate the function call itself, which the Go compiler does, under certain conditions, by replacing the call to a function with the contents of the function. This is known as *inlining* because it brings the body of the function in line with its caller.

Improved optimisation opportunities

Dr. Cliff Click describes inlining as *the* optimisation performed by modern compilers as it forms the basis for optimisations like constant propagation and dead code elimination. In effect, inlining allows the compiler to *see further*, allowing it to observe, in the context that a particular function is being called, logic that can be further simplified or eliminated entirely. As inlining can be applied recursively optimisation decisions can be made not only in the context of each individual function, but also applied to the chain of functions in a call path.

Inlining in action

The effects of inlining can be demonstrated with this small example

```
package main

import "testing"

//go:noinline
func max(a, b int) int {
    if a > b {
        return a
    }
    return b
}

var Result int

func BenchmarkMax(b *testing.B) {
    var r int

    for i := 0; i < b.N; i++ {
        r = max(-1, i)
    }
    Result = r
}
```

Running this benchmark gives the following result:[3]

% **go test -bench=.**
BenchmarkMax-4   530687617   2.24 ns/op

The cost of `max(-1, i)` is around 2.24 nanoseconds on my 2015 MacBook Air. Now let's remove the `//go:noinline` pragma and see the result:

% **go test -bench=.**
BenchmarkMax-4   1000000000   0.514 ns/op

From 2.24 ns to 0.51 ns, or according to `benchstat`, a 78% improvement.

% **benchstat {old,new}.txt**
name   old time/op new time/op delta
Max-4 2.21ns ± 1% 0.49ns ± 6% -77.96% (p=0.000 n=18+19)

Where did these improvements come from?

First, the removal of the function call and associated preamble[4] was a major contributor. Pulling the contents of `max` into its caller reduced the number of instructions executed by the processor and eliminated several branches.

Now the contents of `max` are visible to the compiler as it optimises `BenchmarkMax` it can make some additional improvements. Consider that once `max` is inlined, this is what the body of `BenchmarkMax` looks like to the compiler:

```
func BenchmarkMax(b *testing.B) {
    var r int
    for i := 0; i < b.N; i++ {
        if -1 > i {
            r = -1
        } else {
            r = i
        }
    }
    Result = r
}
```

Running the benchmark again we see our manually inlined version performs as well as the version inlined by the compiler

```
% benchstat {old,new}.txt
name   old time/op  new time/op  delta
Max-4  2.21ns ± 1%  0.48ns ± 3%  -78.14%  (p=0.000 n=18+18)
```

Now the compiler has access to the result of inlining `max` into `BenchmarkMax` it can apply optimisation passes which were not possible before. For example, the compiler has noted that `i` is initialised to `0` and only incremented so any comparison with `i` can assume `i` will never be negative. Thus, the condition `-1 > i` will never be true.[5]

Having proved that `-1 > i` will never be true, the compiler can simplify the code to

```
func BenchmarkMax(b *testing.B) {
    var r int
    for i := 0; i < b.N; i++ {
        if false {
            r = -1
        } else {
            r = i
        }
    }
    Result = r
}
```

and because the branch is now a constant, the compiler can eliminate the unreachable path leaving it with

```
func BenchmarkMax(b *testing.B) {
    var r int
    for i := 0; i < b.N; i++ {
        r = i
    }
    Result = r
}
```

Thus, through inlining and the optimisations it unlocks, the compiler has reduced the expression `r = max(-1, i)` to simply `r = i`.

The limits of inlining

In this article I've discussed, so called, *leaf* inlining; the act of inlining a function at the bottom of a call stack into its direct caller. Inlining is a recursive process, once a function has been inlined into its caller, the compiler may inline the resulting code into *its* caller, as so on. For example, this code

```
func BenchmarkMaxMaxMax(b *testing.B) {
    var r int
    for i := 0; i < b.N; i++ {
        r = max(max(-1, i), max(0, i))
    }
    Result = r
}
```

runs as fast as the previous example as the compiler is able to repeatedly apply the optimisations outlined above to reduce the code to the same `r = i` expression.

In the next article I'll discuss an alternative inlining strategy when the Go compiler wishes to inline a function in the middle of a call stack. Finally I'll discuss the limits that the compiler is prepared to go to to inline code, and which Go constructs are currently beyond its capability.

In Go, a method is just a function with a predefined formal parameter, the receiver. The relative costs of calling a free function vs a invoking a method, assuming that method is not called through an interface, are the same.

Up until Go 1.14 the stack check preamble was also used by the garbage collector to stop the world by setting all active goroutine's stacks to zero, forcing them to trap into the runtime the next time they made a function call. This system was recently replaced with a mechanism which allowed the runtime to pause an goroutine without waiting for it to make a function call.

I'm using the `//go:noinline` pragma to prevent the compiler from inlining `max`. This is because I want to isolate the effects of inlining on `max` rather than disabling optimisations globally with `-gcflags='-l -N'`. I go into detail about the `//go:` comments in this presentation.

You can check this for yourself by comparing the output of `go test -bench=. -gcflags=-S` with and without the `//go:noinline` annotation.

You can check this yourself with the `-gcflags=-d=ssa/prove/debug=on` flag.