DevFest
Siberia 2018

Go inliner: past, present and the future

Backend

Iskander (Alex) Sharipov

Go contributor, open source enthusiast

The only slide

DevFest Siberia 2018

Problem: function calls are slow.
Solution: just inline them.

Fin. Problem solved.

The only slide (not really)

Inlined everything

Now we have **two** problems:

1. Overall performance degraded.
2. Binary became larger.

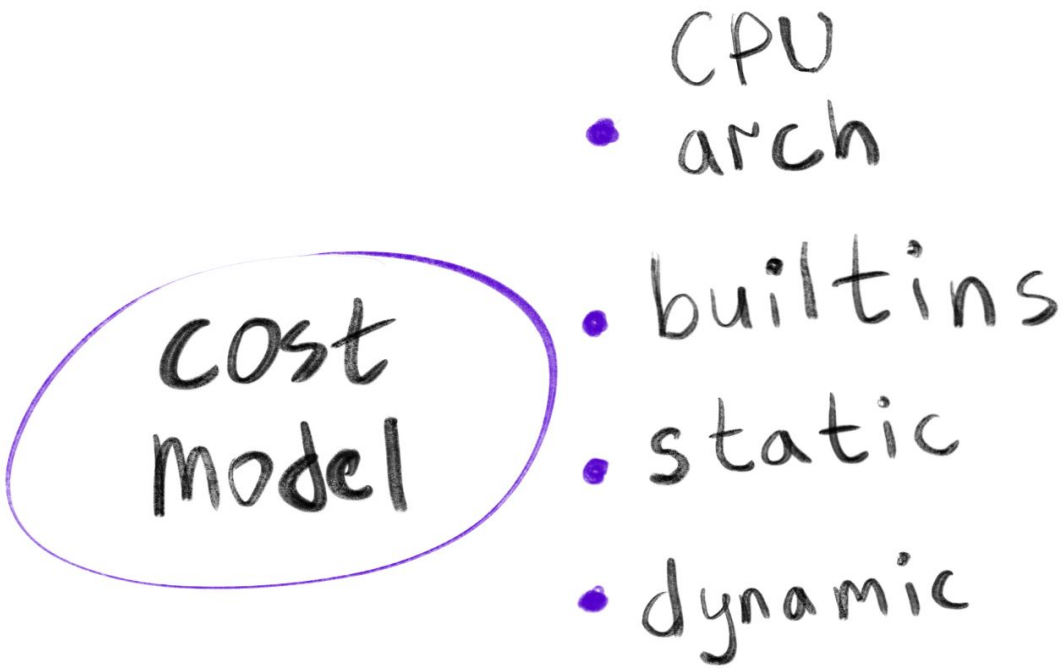
I thought I solved the problem

High-level overview

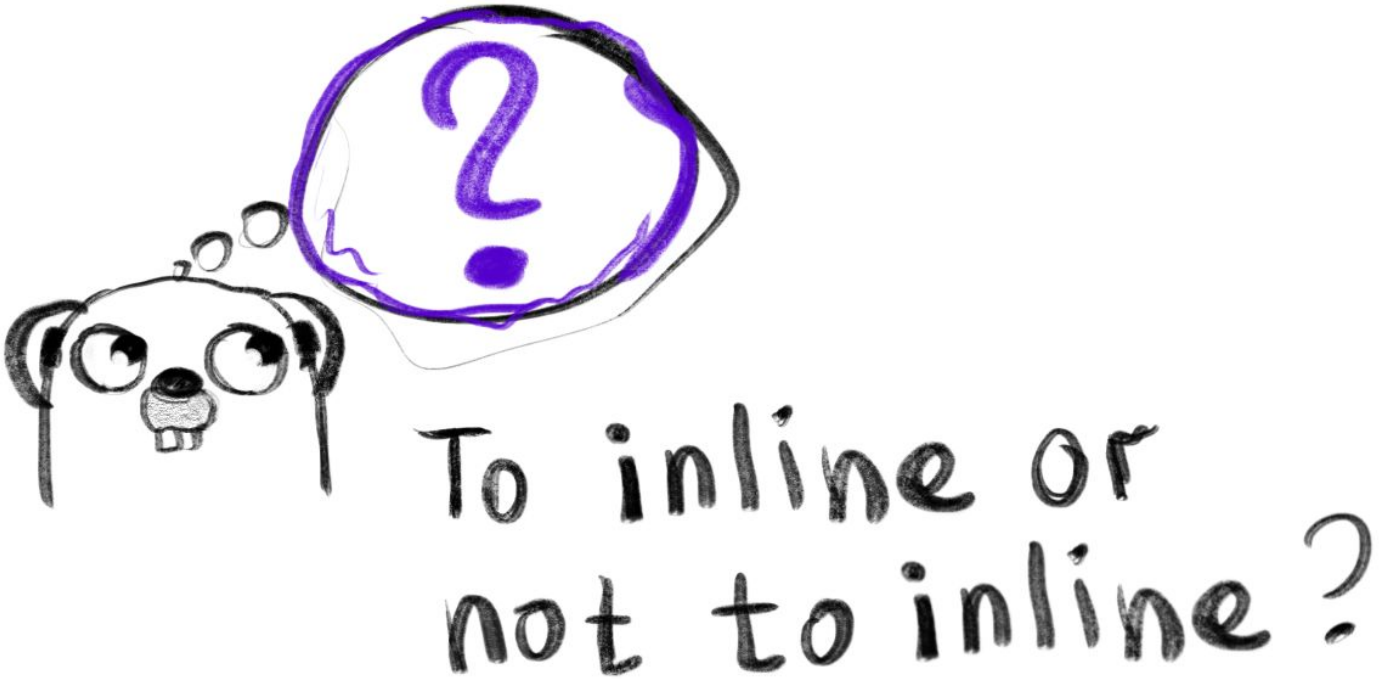
Inlining dichotomy

- No inlining makes code run slow.
- Too much inlining makes code run slow and binaries become bigger.

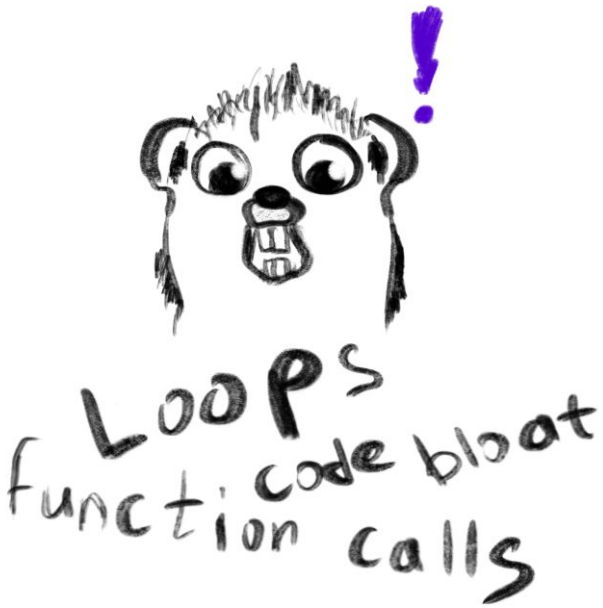
High-level overview



High-level overview



High-level overview



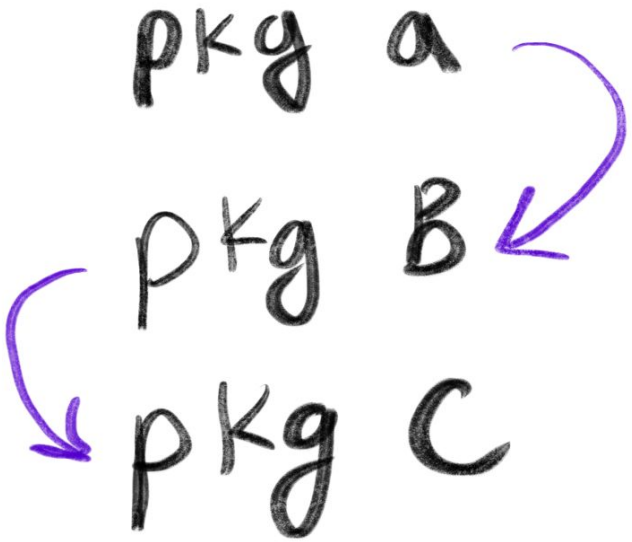
Main parts of the inliner

1) **Cost model.**
   Evaluate the approx “code size” for the function body.
2) **Decision making.**
   Tells if function is inlineable in general or on a call site.
3) **Inlining algorithm and trade-offs handling.**
   How inlining is performed and when it’s not supported.

Inliner: the good parts

(Before we start discussing everything else)

Inliner: the good parts



Compiler can inline function calls between packages. This makes packages that provide convenience wrappers around inlineable functions **almost** free.

Inliner: the good parts


```
func canInlineFuncLitCall() int {
   add1 := func(x int) int {
      return x + 1
   }
   return add1(10) // Inlined
}
// Optimized to return 11
```

Inliner: the good parts

DevFest Siberia 2018

```
func canInlineClosureCall() int {
  x := 10
  add1 := func() int {
    return x + 1
  }
  return add1() // “return 11”
}
```

Inliner: the good parts


```
func canInlineExplicitPanic() {
  panic("can be inlined")
}

// Implicit panics are also
// handled. They can occur in
// slice indexing expressions.
```

Mid-stack inlining (inlining of non-leaf calls)

- We almost have it...
- Still causes significant code bloating. But it gets better.

https://golang.org/issue/19348

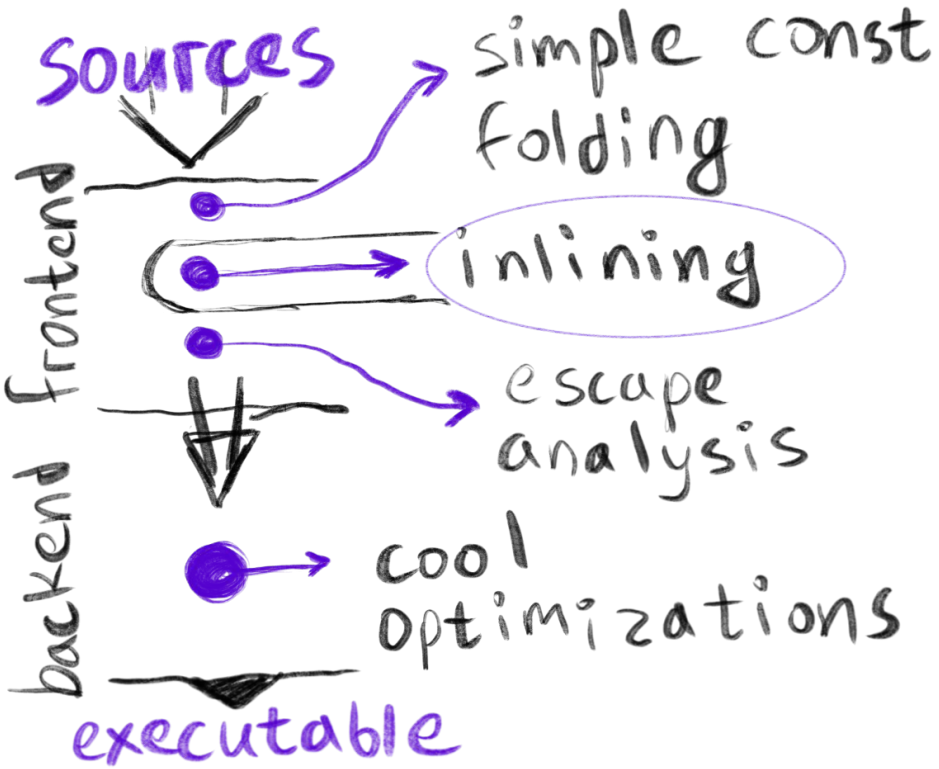
Function “cost” calculation

Cost is a function over syntax

More syntactical elements means higher cost. Sometimes one can re-write code so it has lower cost but operates in the same way.

Function “cost” calculation

Function “cost” calculation



Static vs dynamic cost

The inliner is very static. If cost is calculated and it’s out of the budget, function is never an inlining candidate.

Static approach disadvantage

Even if function can be further optimized or even const-folded with constant arguments, it just won’t happen.

Function “cost” calculation

DevFest Siberia 2018

```
xs = append(xs, 1) // cost=5
x := 1             // cost=5

// Builtin costs are weird.
//
// Budget of 80 with appends
// leads to >1kb of x86 code!
```

The cost of wrapping

Forwarding function does not have any execution time costs, but every inlining increases total cost by introducing extra AST nodes.

Inlining dichotomy

- It looks like there are so many things to improve...
- Yet, most of these paths lead to performance regressions.

https://golang.org/issue/17566

“Extra budget” experiment

Hypothesis 1/2

Inlining is more significant inside loops. Constant arguments also make inlining more eligible.

Hypothesis 2/2

Conditional branch that is not known to be “likely taken” makes inlining less beneficial.

Proposed solution

Increase maximum inlining budget. Budget starts from the base and can be changed through the program control flow.

“Extra budget” experiment


```
func f(x, y int) {
   // Does something smart.
}

// Imagine that f has a cost of
// 95. Base budget is 80, so we
// can’t inline it normally.
```

“Extra budget” experiment

DevFest Siberia 2018

```
// Base budget is 80.
for _, x := range xs { // 90
  for _, y := range ys { // 100
    f(x, y) // Can inline!
  } // budget is 90 again
} // budget is 80 again
```

“Extra budget” experiment

DevFest Siberia 2018

```
for _, x := range xs { // 90
  for _, y := range ys { // 100
    if x + y < 100 { // 90
      f(x, y) // Won't inline
    } // budget is 100 again
  } // budget is 90 again
} // budget is 80 again
```

“Extra budget” experiment

DevFest Siberia 2018

```
// Constant arguments effect.

var x, y int
f(x, y)    // 80, won’t inline
f(10, y)   // 90, won’t inline
f(10, 20) // 100, can inline!
```

Pros & Cons

- Pros: simple to implement, doesn’t make compilation slower, avoids code bloating.
- The real benefits, apart from the microbenchmarks, are not apparent.
  It doesn’t seem to work.

Recent improvements

(The present)

Recent improvements

DevFest Siberia 2018

```
  var total, i int
loop:
  if i == len(xs) {
    return total
  }
  total += xs[i]
  i++
  goto loop
```

Recent improvements

DevFest Siberia 2018

```
func countSum(xs []int) int {
  var total int
  for _, x := range xs {
    total += x
  }
  return total
}
// Inlineable with CL148777.
```

Less inlining in big functions

The budget drops from 80 to 20 for very large functions.
See https://golang.org/cl/125516.

Recent improvements

DevFest Siberia 2018

```
func wrapper(x int) int {
  return nonLeafCall(x)
}
// Inlineable with CL147361.

// Functions with a single non-leaf call
// inside bodies can be inlineable now.
// This is a part of mid-stack enabling
// work that is done by David Chase.
```

Potential improvements

(The future?)

Additional inlining round(s)

Some functions can become trivially inlineable after SSA optimizations. Second inlining round could inline them.

Arg-based specializations

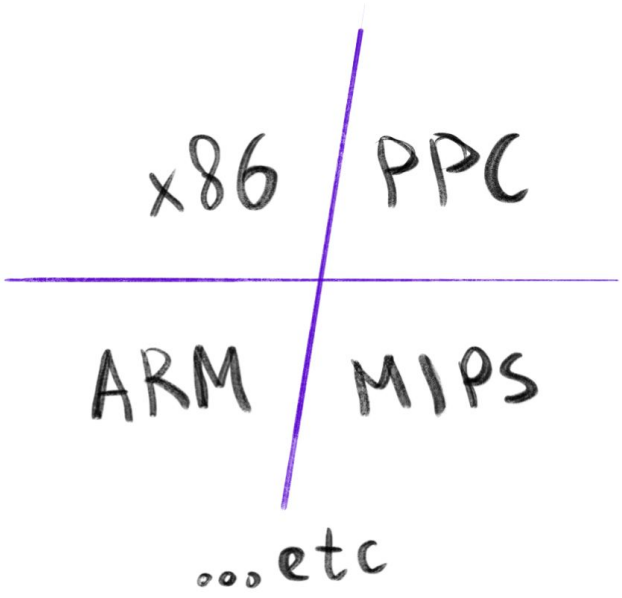
Collect constant argument control flow specialization if it fits the budget. It can solve issues like #27149.

More special-casing for idioms

Constructor functions, for example, can benefit from being inlined, since it can make more created objects stack-allocated.

Fundamental problems

Fundamental problems



Even inside one arch, like AMD64, optimal inliner parameters can vary. Not to mention how different it is for the other arch (instruction cache is one of the concerns).

Fundamental problems

Lack of compile flags demands “best fit defaults”. Compile time is important in Go, we can’t perform expensive operations (and the user can’t ask for that).

Calling convention

If Go calling convention changes from stack-based to register-based we should reconsider some inliner heuristics.

Pragmatic hints

(Don’t follow blindly)

Don’t rely on inlining (if you can)

If possible, avoid even thinking about inlining and how it will affect your application performance. It can be irrelevant.

Write idiomatic Go code

Inlining heuristics may depend on some very widely used Go idioms to choose the right thing.

Manual function specialization

Split big CPU-intensive functions into several smaller ones (inlineable) and call viable specialization on the call site.

Write inlining tests

If it is crucial for some functions to be inlineable, add a test for that.

github.com/Quasilyte/inltest

Pragmatic hints

You can forbid function inlining.

```
//go:noinline
func coldPathFunc() T {
  // Inlineable, but never
  // called from a hot path.
}
```

Pragmatic hints

You can debug inliner decisions.

```
$ go tool compile -m=2 foo.go
cannot inline coldPathFunc: marked go:noinline
```

Also works with “go build”:

```
$ go build -gcflags='-m=2' foo.go
```

Thanks for attention

Iskander (Alex) Sharipov

Go contributor, open source enthusiast

 @quasilyte

 @quasilyte

 @quasilyte

 @quasilyte