Inlining

bright victories and hidden defeats

Me

- Backend developer @ TradingView
- Go developer since 2012
- Community member since 2015
- Meet-up organizer since 2018
- Conference speaker since 2019 :)

Inline expansion

Inlining is

- Embedding function code inside the body of the caller
- Compiler optimization
  - Can be done manually
- First research papers around 1980s
- Present in all major compilers for C/C++/Java/C#/etc
- Budget based, profiled-guided and so on...

Good

- Eliminating call overhead
  - for Go up to 4-7 nanoseconds on modern CPU’s
- Preserves stack and registers
  - no need to pass arguments by stack
- Good instruction cache locality (locality of reference)
- Works well with optimizations like escape analysis

Bad

- Bigger binaries
  - From 7% to 50% and even bigger
- Cache misses
  - Big functions do not fit in CPU cache
- Mysterios interactions with GC and a runtime

A rule of thumb:
Some inlining will improve speed at very minor cost of space, but excess inlining will hurt speed and cost space.

Inlining in Go compiler

History

- Basic inlining since Go 1.0
    - Some basic tests in https://golang.org/test/inline.go
- Implementation is quite simple
    - Most of it in **cmd/compile/internal/gc/inl.go**
- Mid-stack inlining since Go 1.12

Can inline

- Functions with
  - basic operations
  - goto’s (but not for’s)
  - intristics
  - appends
  - map access
  - panic’s
- Closures
- Non-leaf functions/methods (since Go 1.12)

Can’t inline (for now)

- Functions with
    - for’s
    - defer’s
    - select
    - closures
    - type switch
    - go
    - type declarations

Will never inline (probably)

- Functions with
  - recover (need a frame pointer)
  - no body
- `Funtime.getcaller`
- Functions implemented in assembly
- Functions marked with “`go:noinline`” and so on...

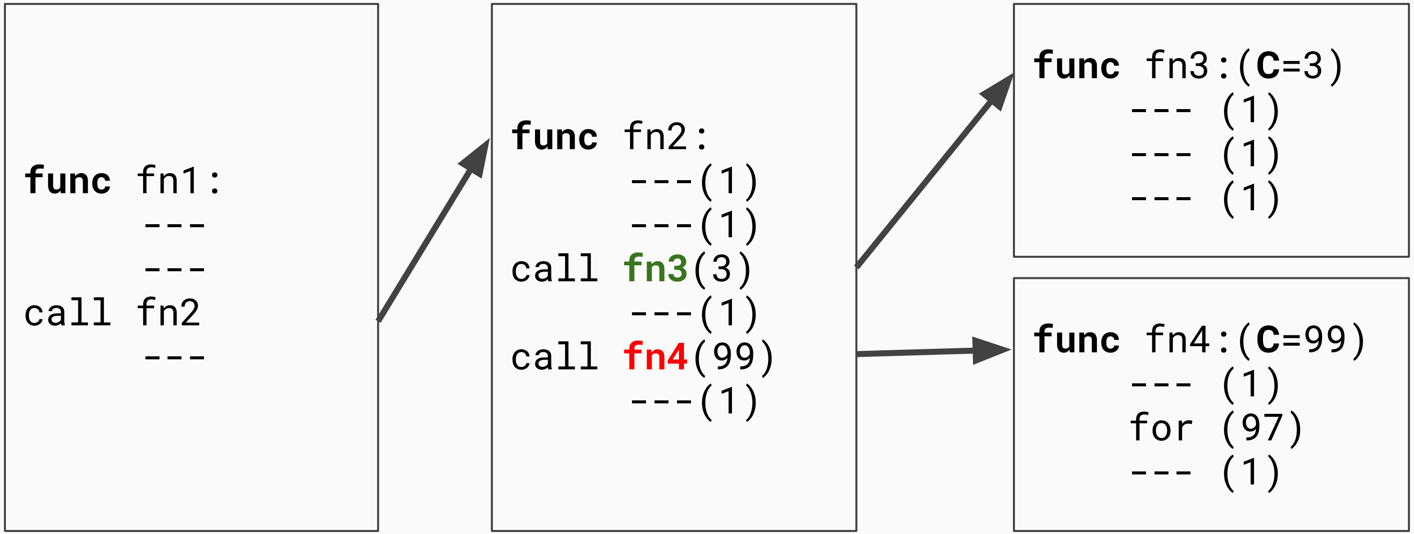
How it works

How it works

- Simple cost-based model
- Every function has a
  - Budget
  - Cost
- Budget defines how much can be inlined inside current function
- Cost defines if the current function can be inlined (and how much it will cost)

How it works

Budget = 80, **C** - Cost, **Can inline**, **Can't inline**



```
func fn1:
     ---
     ---
call fn2
     ---
```

```
func fn2:
     ---(1)
     ---(1)
call fn3(3)
     ---(1)
call fn4(99)
     ---(1)
```

```
func fn3:(C=3)
    --- (1)
    --- (1)
    --- (1)
```

```
func fn4:(C=99)
    --- (1)
    for (97)
    --- (1)
```

Possible improvements:

- Inline for-loops
  - https://github.com/golang/go/issues/14768
- Inline defer
  - https://github.com/golang/go/issues/14939
- Improve inlining cost model
  - https://github.com/golang/go/issues/17566

Quiz time!

Will this exit?

```go
package main

import (
	"runtime"
	"sync/atomic"
)

var (
	variable uint64
)

func main() {
	runtime.GOMAXPROCS(1)
	go func() {
		for {
			atomic.AddUint64(&variable, 1)
		}
	}()
	runtime.Gosched()
}
```

Will this exit?

```
package main

import (
	"runtime"
	"sync/atomic"
)

var (
	variable uint64
)

func main() {
	runtime.GOMAXPROCS(1)
	go func() {
		for {
			atomic.AddUint64(&variable, 1)
		}
	}()
	runtime.Gosched()
}
```

Answer: **No**

Program exited: process took too long.

Will this exit?

```go
package main

import (
        "runtime"
        "sync"
)

var (
        mx       sync.Mutex
        variable uint64
)

func main() {
        runtime.GOMAXPROCS(1)
        go func() {
                for {
                        mx.Lock()
                        variable++
                        mx.Unlock()
                }
        }()
        runtime.Gosched()
}
```

Will this exit?

```go
package main

import (
	"runtime"
	"sync"
)

var (
	mx       sync.Mutex
	variable uint64
)

func main() {
	runtime.GOMAXPROCS(1)
	go func() {
		for {
			mx.Lock()
			variable++
			mx.Unlock()
		}
	}()
	runtime.Gosched()
}
```

Answer: **No**

Program exited: process took too long.

But why?

Safe-points!

Safe-points

- Currently (as Go 1.13) runtime can only stop goroutine’s at safe-points
- Safe points are placed through the resulting code by the compiler
  - Most of them are located at the function’s prologue
- Runtime can’t continue GC before all goroutines reach safe-points
- It can’t switch them too

Will this exit?

```
package main

import (
    "runtime"
    "sync"
)

var (
    mx       sync.Mutex
    variable uint64
)

func main() {
    runtime.GOMAXPROCS(1)
    go func() {
        for {
            mx.Lock()
            variable++
            mx.Unlock()
        }
    }()
    runtime.Gosched()
}
```

Answer: **No (because it’s a deadlock)**

Problems

- Inlining can result in bizarre dead-locks and live-locks
- Can be solved with non-cooperative goroutine preemption
    - https://github.com/golang/go/issues/24543

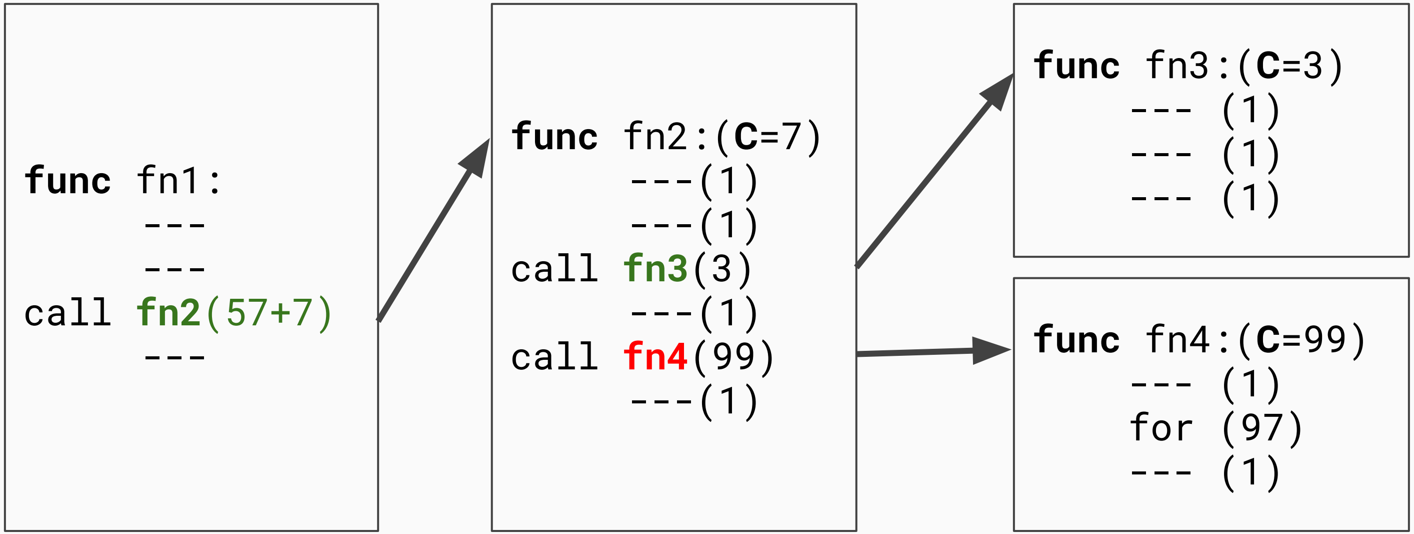
Mid-stack inlining

Mid-stack inlining

- First talks ~ 2016
- Design doc in 2017
    - https://golang.org/design/19348-midstack-inlining
- Enabled behind the flag (`-gcflag=-l4`) since 2017
- Main problem: stack frames
    - Runtime must know where current code executes
        - For stacktraces/panics/callers
- Fully enabled in **Go 1.12**

How it works (since Go 1.12)

Budget = 80, **Non-leaf call cost** = 57, **C** - Cost, **Can inline**, **Can't inline**



```
func fn1:
     ---
     ---
call fn2(57+7)
     ---
```

```
func fn2:(C=7)
        ---(1)
        ---(1)
call fn3(3)
        ---(1)
call fn4(99)
        ---(1)
```

```
func fn3:(C=3)
     --- (1)
     --- (1)
     --- (1)
```

```
func fn4:(C=99)
     --- (1)
     for (97)
     --- (1)
```

Will this exit?

```go
package main

import (
	"runtime"
	"sync"
)

var (
	mx       sync.Mutex
	variable uint64
)

func main() {
	runtime.GOMAXPROCS(1)
	go func() {
		for {
			mx.Lock()
			variable++
			mx.Unlock()
		}
	}()
	runtime.Gosched()
}
```

Answer: **No**

mx.Lock/Unlock were **inlined**

Optimizations!

Simple code

```
package main

import "math"

var GlobalArray [65535]int

func ModifyArrayOnIntMax(v uint64) {
	if v > math.MaxInt64 {
		for i := 0; i < 65535; i++ {
			GlobalArray[i]++
		}
	}
}
```

Simple code

```
package main

import "math"

var GlobalArray [65535]int

func ModifyArrayOnIntMax(v uint64) {
    if v > math.MaxInt64 {
        for i := 0; i < 65535; i++ {
            GlobalArray[i]++
        }
    }
}
```

```
BenchmarkModifyArrayOnIntMax-8        692112469
             1.67 ns/op
BenchmarkModifyArrayOnIntMax-8        724745390
             1.64 ns/op
BenchmarkModifyArrayOnIntMax-8        697325808
             1.70 ns/op
BenchmarkModifyArrayOnIntMax-8        710092806
             1.62 ns/op
BenchmarkModifyArrayOnIntMax-8        741783656
             1.62 ns/op
```

Average ~ **1.60ns**

Sample code

```go
package main

import "math"

var GlobalArray [65535]int

func ModifyArrayOnIntMaxV2(v uint64) {
	if v <= math.MaxInt64 {
		return
	}

	modifyArrayOnIntMaxV2()
}

func modifyArrayOnIntMaxV2() {
	for i := 0; i < 65535; i++ {
		GlobalArray[i]++
	}
}
```

Sample code

```go
package main

import "math"

var GlobalArray [65535]int

func ModifyArrayOnIntMaxV2(v uint64) {
	if v <= math.MaxInt64 {
		return
	}

	modifyArrayOnIntMaxV2()
}

func modifyArrayOnIntMaxV2() {
	for i := 0; i < 65535; i++ {
		GlobalArray[i]++
	}
}
```

```
BenchmarkModifyArrayOnIntMaxV2-8
1000000000              0.270 ns/op
BenchmarkModifyArrayOnIntMaxV2-8
1000000000              0.273 ns/op
BenchmarkModifyArrayOnIntMaxV2-8
1000000000              0.272 ns/op
BenchmarkModifyArrayOnIntMaxV2-8
1000000000              0.269 ns/op
BenchmarkModifyArrayOnIntMaxV2-8
1000000000              0.282 ns/op
```

Average ~ **0.273ns (x6 speedup!)**

Function outlining

Function outlining

- Moving parts of functions into the parent to enable other optimizations.
- For example - compiler can inline the parent function containing hot paths

More optimizations!

Simple code

```go
package main

func AllocateConstantSlice(v int) []int
{
	slc := make([]int, 1024)
	for i := range slc {
		slc[i] = v
	}

	return slc
}
```

Simple code

```go
package main

func AllocateConstantSliceV2(v int) []int {
	slc := make([]int, 1024)
	allocateConstantSliceV2(v, slc)
	return slc
}

func allocateConstantSliceV2(v int, slc []int) {
	for i := range slc {
		slc[i] = v
	}
}
```

```
BenchmarkAllocateConstantSliceV2-8
2864816
413 ns/op
0 B/op
0 allocs/op
```

Credits to:
Filippo Valsorda(@FiloSottile)

Takeaways

- Compiler is your friend
- Use your compiler
- Know your compiler
- Improve your compiler
- Make your compiler 😎

THANK YOU!