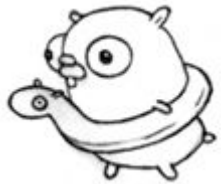
Mid-stack inlining in the Go compiler

David Lazar
mentored by Austin Clements



What is inlining?

Inlining replaces a call with the body of the function:

```
func Hypot(x, y float64) {
    return Sqrt(x*x + y*y)
}

for x := range xs {
    s += Hypot(x, C)
}
```

```
for x := range xs {
    s += Sqrt(x*x + C*C)
}
```

Why do inlining?

Avoids call overhead:

```
for x := range xs {
     s += Sqrt(x*x + C*C)
}
```

Enables other optimizations, like Loop-Invariant Code Motion:

```
Z := C*C
for x := range xs {
     s += Sqrt(x*x + Z)
}
```

Challenge: stack traces

If Sqrt panics because its input is negative, we should see a complete stack trace, even if **Hypot** is inlined:

```
math.Sqrt(0xbff0000000000000, 0xc420000180)
    /go/src/math/sqrt.go:53 +0x64

math.Hypot(...)
    /go/src/math/hypot.go:108

main.main()
    /go/src/app/main.go:77 +0x2f
```

Challenge: stack traces

Without a call, Go currently can’t print a frame for **Hypot**:

```
math.Sqrt(0xbff0000000000000, 0xc420000180)
    /go/src/math/sqrt.go:53 +0x64

math.Hypot(...)
    /go/src/math/hypot.go:108

main.main()
    /go/src/app/main.go:77 +0x2f
```

Go's current solution:
don’t inline mid-stack calls

Mid-stack inlining:
inlining functions that call other functions

9% improvement on benchmarks

Contributions

Modified the compiler and runtime to generate accurate stack traces with mid-stack inlining.

Made runtime.Caller(s) work with inlining.

Fixed several bugs where runtime assumed no inlining.

Performance evaluation

Running example

```
type Point struct { X, Y *big.Int }

func (p *Point) Flip(d bool) {
    if d {
        p.X.Neg()
    } else {
        p.Y.Neg()
    }
}

func (z *Int) Neg() {
    z.neg = !z.neg
}
```

```
func main() {
    var p Point
    p.Flip(*direction)
}
```

Neg is a leaf, so compiler inlines it

```
type Point struct { X, Y *big.Int }

func (p *Point) Flip(d bool) {
    if d {
        p.X.neg = !p.X.neg
    } else {
        p.Y.neg = !p.Y.neg
    }
}

func (z *Int) Neg() {
    z.neg = !z.neg
}
```

```
func main() {
    var p Point
    p.Flip(*direction)
}
```

Now Flip is a leaf, so compiler inlines it

```
type Point struct { X, Y *big.Int }

func (p *Point) Flip(d bool) {
    if d {
        p.X.neg = !p.X.neg
    } else {
        p.Y.neg = !p.Y.neg
    }
}

func (z *Int) Neg() {
    z.neg = !z.neg
}
```

```
func main() {
    var p Point
    d := *direction
    if d {
        p.X.neg = !p.X.neg
    } else {
        p.Y.neg = !p.Y.neg
    }
}
```

Currently, stack traces are incomplete even without mid-stack inlining!

```
panic: nil pointer dereference

main.main()
    /go/src/app/main.go:12 +0x2a
```

```
func main() {
    var p Point
    d := *direction
    if d {
        p.X.neg = !p.X.neg
    } else {
        p.Y.neg = !p.Y.neg
    }
}
```

Our approach

Step 1: extend AST position information into a tree of inlined call positions

Step 2: export the inlining tree via runtime symbol tables

Step 3: use inlining tree at runtime to expand stack traces and caller information

Same example with line numbers

```
type Point struct { X, Y *big.Int }


41:  func (p *Point) Flip(d bool) {
42:       if d {
43:            p.X.Neg()
44:       } else {
45:            p.Y.Neg()
46:       }
47:  }

97:  func (z *Int) Neg() {
98:       z.neg = !z.neg
99:  }
```

```
10: func main() {
11:      var p Point
12:      p.Flip(*direction)
13: }
```

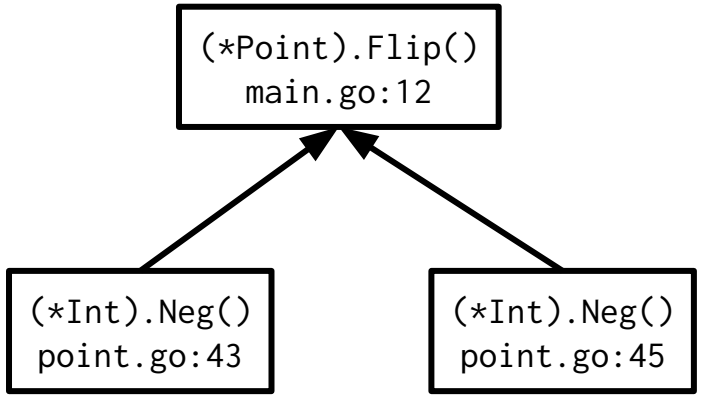
Modified inliner to copy AST positions

```
type Point struct { X, Y *big.Int }

41:  func (p *Point) Flip(d bool) {
42:      if d {
43:          p.X.Neg()
44:      } else {
45:          p.Y.Neg()
46:      }
47:  }

97:  func (z *Int) Neg() {
98:      z.neg = !z.neg
99:  }
```

```
10:  func main() {
11:      var p Point
12:      d := *direction
42:      if d {
98:          p.X.neg = !p.X.neg
44:      } else {
98:          p.Y.neg = !p.Y.neg
46:      }
13:  }
```

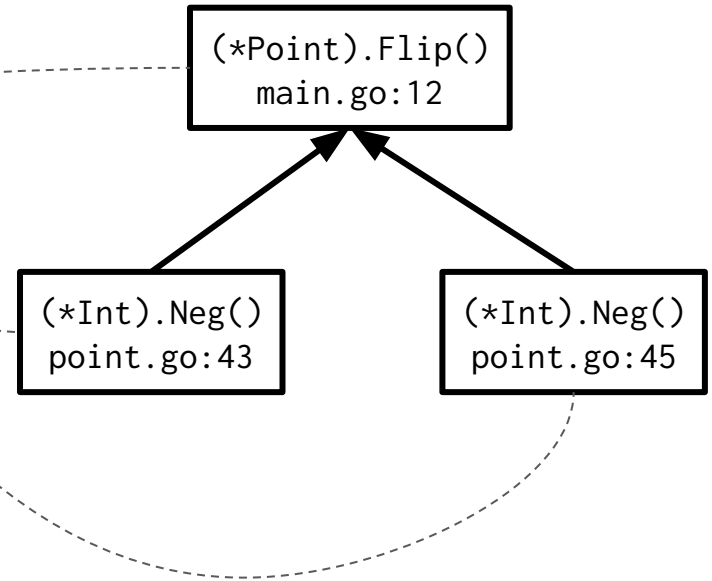
Compiler maintains a tree of inlined calls

```
func main() {
    var p Point
    d := *direction
    if d {
        p.X.neg = !p.X.neg
    } else {
        p.Y.neg = !p.Y.neg
    }
}
```



Every AST node maps to a node in the inlining tree

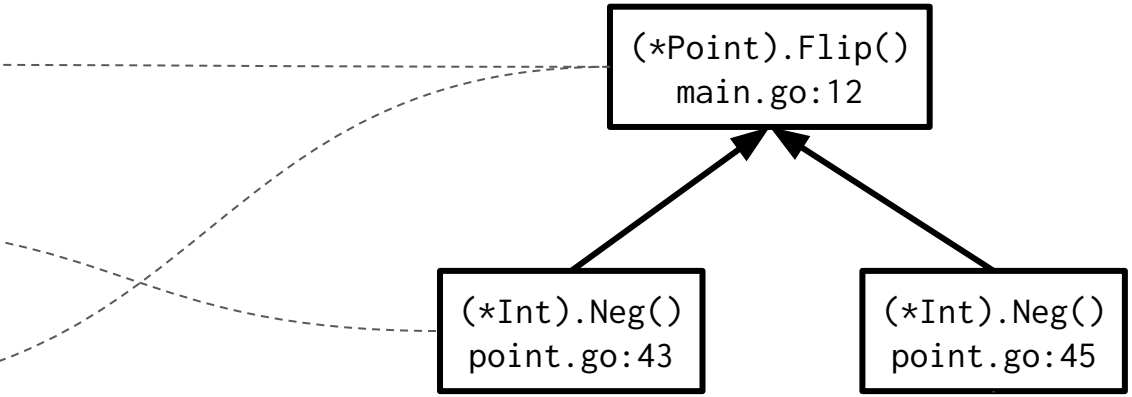
```
func main() {
    var p Point
    d := *direction
    if d {
        p.X.neg = !p.X.neg
    } else {
        p.Y.neg = !p.Y.neg
    }
}
```



For any AST node, walk the path up the tree to generate an accurate stack trace!

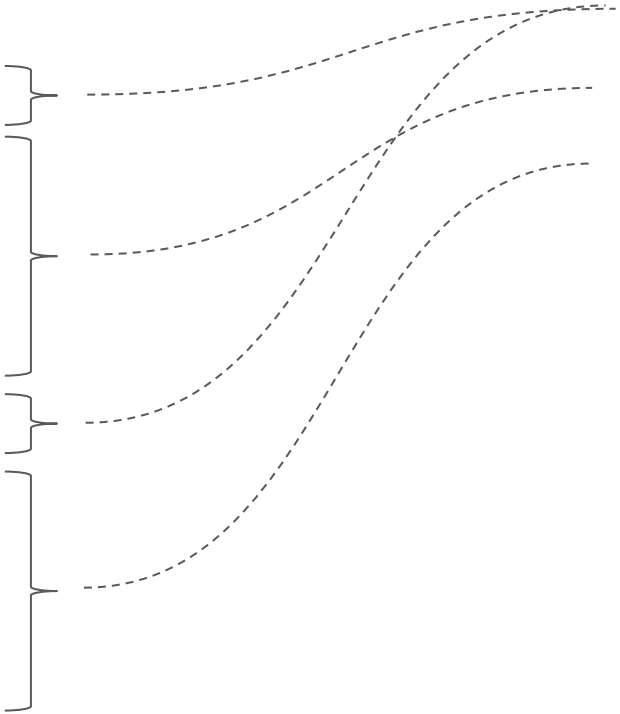
Mapping is preserved as AST is lowered

```
func main():
  MOVQ
  MOVQ
  JEQ
  MOVQ
  MOVB
  XORL
  MOVB
  JMP
  MOVQ
  MOVB
  XORL
  MOVB
```



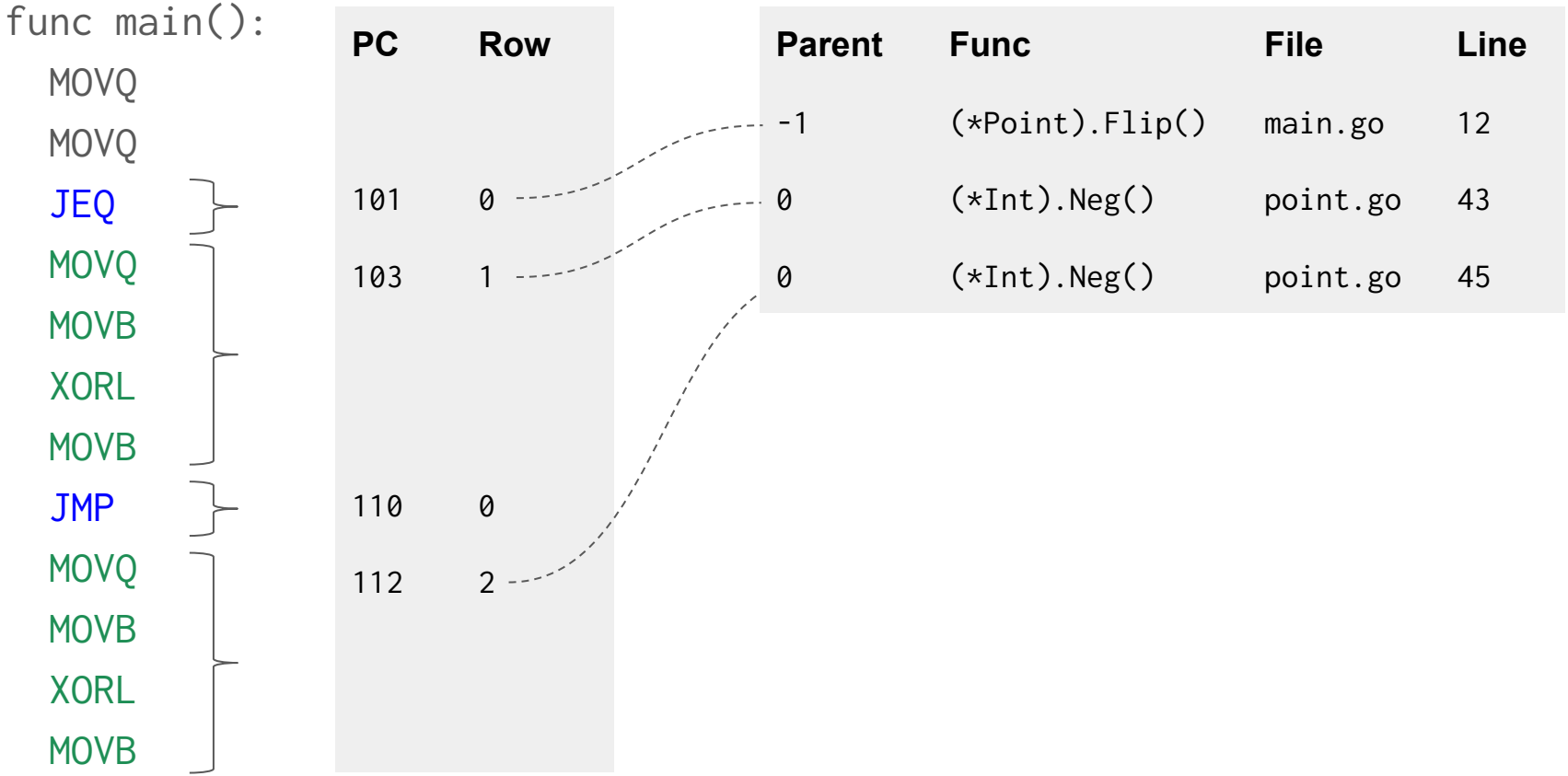
Compiler encodes inlining tree as a table

```
func main():
  MOVQ
  MOVQ
  JEQ
  MOVQ
  MOVB
  XORL
  MOVB
  JMP
  MOVQ
  MOVB
  XORL
  MOVB
```



Parent	Func	File	Line
-1	(*Point).Flip()	main.go	12
0	(*Int).Neg()	point.go	43
0	(*Int).Neg()	point.go	45

PC-value table maps each PC to a row in the inlining table



Linker compactly encodes inlining tables in binary

```
func main():
  MOVQ
  MOVQ
  JEQ
  MOVQ
  MOVB
  XORL
  MOVB
  JMP
  MOVQ
  MOVB
  XORL
  MOVB
```

PC	Row
101	0
103	1
110	0
112	2

Parent	Func	File	Line
-1	(*Point).Flip()	main.go	12
0	(*Int).Neg()	point.go	43
0	(*Int).Neg()	point.go	45

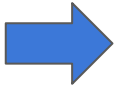
Parent	Func	File	Line
-1	72	4	12
0	104	5	43
0	104	5	45

Runtime reads tables from PCDATA and FUNCDATA

```
func main():
    MOVQ
    MOVQ
    JEQ
    MOVQ
    MOVB
    XORL
    MOVB
    JMP
    MOVQ
    MOVB
    XORL
    MOVB
```

PC	Row
101	0
103	1
110	0
112	2

Parent	Func	File	Line
-1	72	4	12
0	104	5	43
0	104	5	45



```
type Func struct {
    Pcdata   [][]byte
    Funcdata [][]byte
    ...
}
```

Runtime

Runtime generates accurate stack traces using the inlining tables

```
panic: nil pointer dereference

big.(*Int).Neg(...)
     /go/src/math/big/int.go:98
main.(*Point).Flip(...)
     /go/src/point/point.go:43
main.main()
     /go/src/app/main.go:12 +0x2a
```

```
func main() {
    var p Point
    d := *direction
    if d {
        p.X.neg = !p.X.neg
    } else {
        p.Y.neg = !p.Y.neg
    }
}
```

Limitation: no arguments for inlined calls

```
panic: nil pointer dereference

big.(*Int).Neg(...)
     /go/src/math/big/int.go:98
main.(*Point).Flip(...)
     /go/src/point/point.go:43
main.main()
     /go/src/app/main.go:12 +0x2a
```

Runtime gets arguments by assuming a certain stack layout, but there’s no stack frame for inlined calls!

Another form of stack trace: runtime.Callers

Applications can build their own stack traces using runtime.Callers:

```
func Callers(skip int, pcs []uintptr) int
```

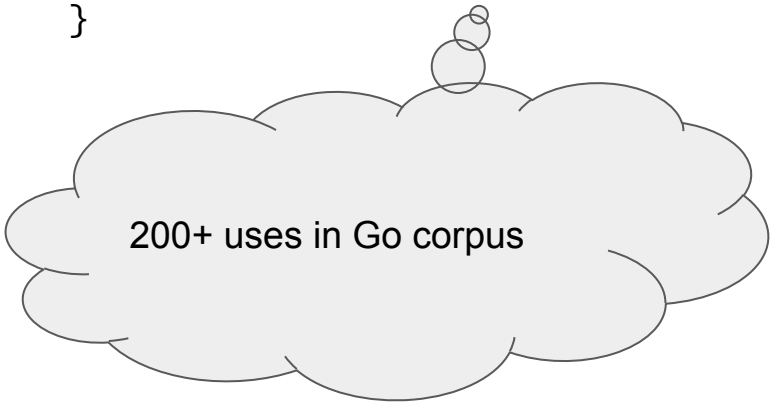
Common pattern is to iterate over the slice of PCs to create a stack trace:

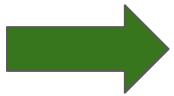
```
pcs := make([]uintptr, 32)
n := runtime.Callers(skip, pcs)
for i := 0; i < n; i++ {
    fn := runtime.FuncForPC(pcs[i])
    log.Println(fn.Name())
}
```

Problem: each PC represents multiple calls! This prints just the outermost functions.

Iterating over PCs is **deprecated**

```
pcs := make([]uintptr, 32)
n := runtime.Callers(skip, pcs)
for i := 0; i < n; i++ {
     fn := runtime.FuncForPC(pcs[i])
     log.Println(fn.Name())
}
```





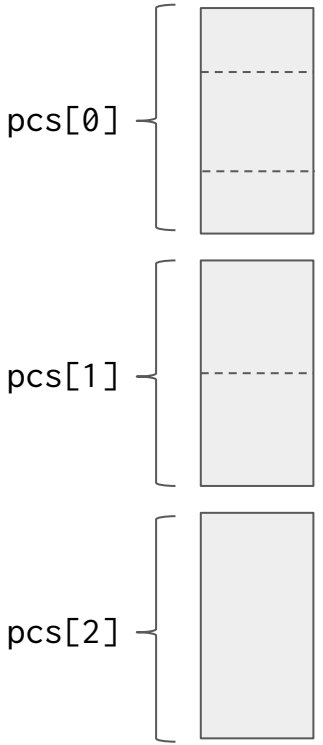
```
pcs := make([]uintptr, 32)
runtime.Callers(skip, pcs)
frames := runtime.CallersFrames(pcs)
for {
     f, more := frames.Next()
     log.Println(f.Func)
     if !more {
          break
     }
}
```

Partially skipped PC

Recall, runtime.Callers returns a slice of PCs:

```
func Callers(skip int, pcs []uintptr) int
```

Suppose the PC at pcs[0] represents 3 calls.



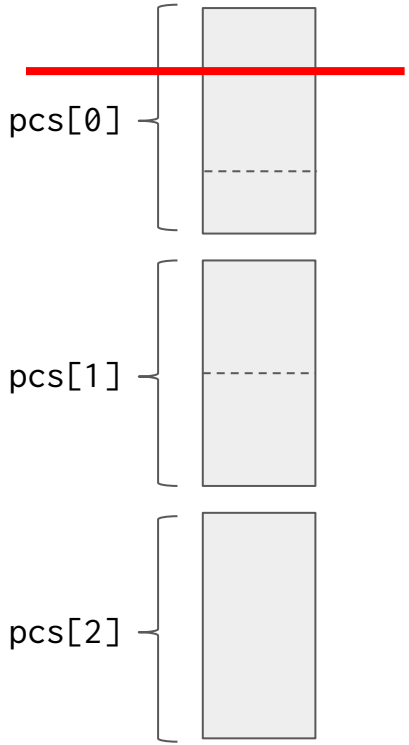
Partially skipped PC

Recall, runtime.Callers returns a slice of PCs:

```
func Callers(skip int, pcs []uintptr) int
```

Suppose the PC at pcs[0] represents 3 calls.

What if skip=1? We can't change the return type.



Partially skipped PC

Recall, runtime.Callers returns a slice of PCs:

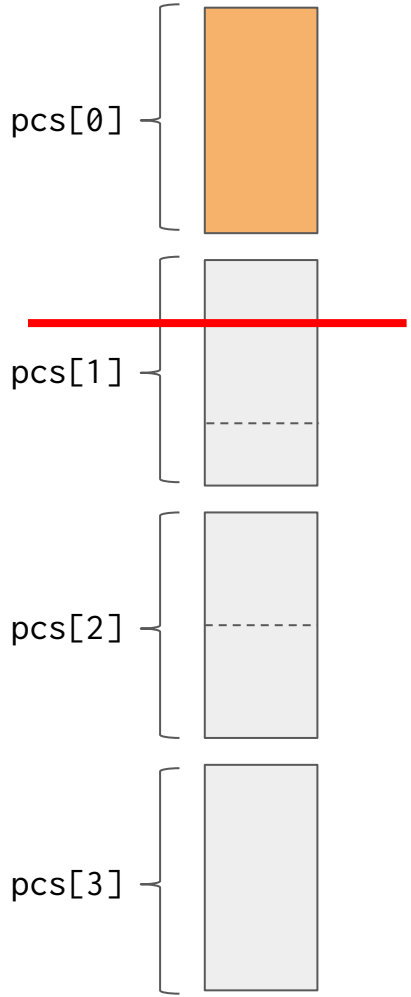
```
func Callers(skip int, pcs []uintptr) int
```

Suppose the PC at pcs[0] represents 3 calls.

What if skip=1? We can't change the return type.

Solution: encode skip as PC into empty function:

```
func skipPleaseUseCallersFrames()
```



Evaluation

How much does mid-stack inlining improve performance?

What is the impact on binary size?

What is the performance impact on Google? (omitted)

Performance: 9% faster on Go1 benchmarks

BinaryTree17-4	2.50s ± 2%	2.52s ± 1%	~	(p=0.421 n=5+5)
FmtFprintfString-4	81.2ns ± 5%	61.8ns ± 1%	-23.90%	(p=0.008 n=5+5)
FmtFprintfFloat-4	236ns ± 3%	214ns ± 0%	-9.08%	(p=0.008 n=5+5)
GobDecode-4	6.51ms ± 1%	5.97ms ± 3%	-8.19%	(p=0.008 n=5+5)
GobEncode-4	5.53ms ± 2%	4.59ms ± 1%	-16.88%	(p=0.008 n=5+5)
Gzip-4	238ms ± 0%	240ms ± 0%	+0.71%	(p=0.008 n=5+5)
Gunzip-4	34.7ms ± 0%	32.1ms ± 0%	-7.41%	(p=0.008 n=5+5)
HTTPClientServer-4	49.5µs ± 1%	48.9µs ± 0%	-1.09%	(p=0.016 n=5+4)
JSONEncode-4	16.5ms ±10%	13.6ms ± 1%	-17.95%	(p=0.008 n=5+5)
JSONDecode-4	53.3ms ± 1%	51.3ms ± 2%	-3.80%	(p=0.008 n=5+5)
Mandelbrot200-4	3.40ms ± 0%	3.41ms ± 0%	+0.32%	(p=0.016 n=5+4)
GoParse-4	3.20ms ± 4%	2.98ms ± 2%	-7.00%	(p=0.008 n=5+5)
RegexpMatchMedium_1K-4	37.3µs ± 3%	34.4µs ± 0%	-7.72%	(p=0.016 n=5+4)
RegexpMatchHard_1K-4	56.7µs ± 2%	53.2µs ± 0%	-6.10%	(p=0.008 n=5+5)
Revcomp-4	422ms ± 1%	415ms ± 1%	-1.64%	(p=0.008 n=5+5)
Template-4	53.1ms ± 2%	48.6ms ± 2%	-8.57%	(p=0.008 n=5+5)
TimeFormat-4	335ns ±11%	279ns ± 0%	-16.82%	(p=0.016 n=5+4)
[Geo mean]	48.4µs	44.0µs	**-9.15%**	

Binary size

+**4%** just to fix stack traces without mid-stack inlining

+**15%** with mid-stack inlining.

- There’s some room to compress the inlining tree.
- Go-specific strip tool for mobile apps.
- Better inlining heuristics.

Project status

9 CLs (submitted)

Fix inlining variadics; export linknames; src.Pos

6 CLs (submitted)

Add inlining tables; fix stack traces

8 CLs (to be mailed)

Fix runtime.Caller(s); make tests pass with mid-stack inlining

Future work

Go 1.9

- Update DWARF tables with inlining info
- Better inlining heuristics
- Support for debug/gosym
- Make mid-stack inlining the default

Go 1.10+

- Arguments in stack traces for inlined calls

Conclusion

With complete stack traces we can turn on mid-stack inlining in 1.9

9% improvement on Go1 benchmarks

Avoid **FuncForPC**, use **CallersFrames**

Follow along: golang.org/issue/19348

Contact: `lazard@{csail.mit.edu,golang.org}`