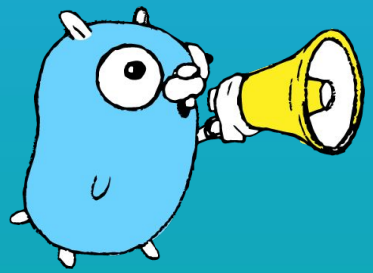
Bengaluru, November 12, 2020

Common patterns for Bounds Check Elimination

Agniva De Sarker

Mattermost

@agnivade



Agenda

SECTION ONE

What are bounds checks ?

```
a[i] = n

*a + i = n

if i > len(a) {
  panic(“index out of range”)
}
*a + i = n
```

Bounds Check

In a nutshell

- For every a[i], the compiler inserts a check to verify 0 <= i < len(a).
- Can be quite painful for performance sensitive apps.

Follow-up thoughts

- Do we really need a bounds check ? Can a program function without it ?
- Yes.
- But ...

So then what ?

- Do we sacrifice performance for safety ?
- Can we have both ?

Enter Bounds Check Elimination

- The compiler analyzes the code and removes bound checks where it can prove that the index is within bounds.

```
for i:=0; i < len(a); i++ {
    _ = a[i] // no bounds check !
}
```

- But what about

```
for i:=0; i < 5; i++ {
    _ = a[i] // bounds check needed
}
```

- The compiler cannot remove it. But we can help!

How to see bound checks

```
$go build -gcflags=”-d=ssa/check_bce/debug=1” foo.go
OR
$go tool compile -d=ssa/check_bce/debug=1 foo.go

foo.go:26:7: Found IsInBounds
foo.go:48:22: Found IsSliceInBounds
```

SECTION TWO

Peeking under the hood

Consider a code like this

```
package main

func foo(b []byte, n int) {
    b[n] = 9
}


$go tool compile -S foo.go
```

Consider a code like this

```
00000 (foo.go:3) TEXT "".foo(SB), NOSPLIT|ABIInternal,$8-32
00000 (foo.go:3) SUBQ    $8, SP
00004 (foo.go:3) MOVQ    BP, (SP)
00008 (foo.go:3) LEAQ    (SP), BP
00012 (foo.go:4) MOVQ    "".n+40(SP), AX // AX=n
00017 (foo.go:4) MOVQ    "".b+24(SP), CX // CX=len(b)
00022 (foo.go:4) CMPQ    AX, CX          // u(n)-u(len(b))=x
00025 (foo.go:4) JCC     45              // J45 if x>=0
00027 (foo.go:4) MOVQ    "".b+16(SP), CX // CX=b
00032 (foo.go:4) MOVB    $9, (CX)(AX*1)  // *b[n]=9
00036 (foo.go:5) MOVQ    (SP), BP
00040 (foo.go:5) ADDQ    $8, SP
00044 (foo.go:5) RET
00045 (foo.go:4) CALL    runtime.panicindex(SB)// panic !
00050 (foo.go:4) UNDEF
```

```
func foo(b []byte, n int) {

    b[n] = 9

}
```

SECTION THREE

Common patterns

Pattern 1

Loop upper bound is known

```
func bce(b []byte, n int) {
    for i := 0; i < n; i++ {
        b[i] = 9
    }
}
```

```
$go tool compile -d=s/c/d=1 foo.go
foo.go:5:8: Found IsInBounds
```

Pattern 1

Loop upper bound is known

```
func bce(b []byte, n int) {
    _ = b[n-1]
    for i := 0; i < n; i++ {
        b[i] = 9
    }
}
```

```
$go tool compile -d=s/c/d=1 foo.go
foo.go:4:8: Found IsInBounds
```

Pattern 2

Contiguous slice access

```
func bce(b []byte, v uint32)
{
    b[0] = byte(v >> 24)
    b[1] = byte(v >> 16)
    b[2] = byte(v >> 8)
    b[3] = byte(v)
}
```

```
$go tool compile -d=s/c/d=1 foo.go
foo.go:5:7: Found IsInBounds
foo.go:6:7: Found IsInBounds
foo.go:7:7: Found IsInBounds
foo.go:8:7: Found IsInBounds
```

Pattern 2

Contiguous slice access

```
func bce(b []byte, v uint32) {
    _ = b[3]
    b[0] = byte(v >> 24)
    b[1] = byte(v >> 16)
    b[2] = byte(v >> 8)
    b[3] = byte(v)
}
```

```
$go tool compile -d=s/c/d=1 foo.go
foo.go:4:7: Found IsInBounds
```

Pattern 3

Slice accesses offset from a base index

```
func bce(b []byte, n int) {
    b[n+0] = byte(1)
    b[n+1] = byte(1 >> 8)
    b[n+2] = byte(1 >> 16)
    b[n+3] = byte(1 >> 24)
    b[n+4] = byte(1 >> 32)
    b[n+5] = byte(1 >> 40)
}
```

```
$go tool compile -d=s/c/d=1 foo.go
foo.go:4:9: Found IsInBounds
foo.go:5:9: Found IsInBounds
foo.go:6:9: Found IsInBounds
foo.go:7:9: Found IsInBounds
foo.go:8:9: Found IsInBounds
foo.go:9:9: Found IsInBounds
```

Pattern 3

Slice accesses offset from a base index

```
func bce(b []byte, n int) {
    _ = b[n+5]
    b[n+0] = byte(1)
    b[n+1] = byte(1 >> 8)
    b[n+2] = byte(1 >> 16)
    b[n+3] = byte(1 >> 24)
    b[n+4] = byte(1 >> 32)
    b[n+5] = byte(1 >> 40)
}
```

```
$go tool compile -d=s/c/d=1 foo.go
foo.go:4:9: Found IsInBounds
foo.go:5:9: Found IsInBounds
foo.go:6:9: Found IsInBounds
foo.go:7:9: Found IsInBounds
foo.go:8:9: Found IsInBounds
foo.go:9:9: Found IsInBounds
foo.go:10:9: Found IsInBounds
```

Pattern 3

Slice accesses offset from a base index

```
func bce(b []byte, n int) {
    b = b[n : n+6]
    b[0] = byte(1)
    b[1] = byte(1 >> 8)
    b[2] = byte(1 >> 16)
    b[3] = byte(1 >> 24)
    b[4] = byte(1 >> 32)
    b[5] = byte(1 >> 40)
}
```

```
$go tool compile -d=s/c/d=1 foo.go
foo.go:4:9: Found IsSliceInBounds
```

Pattern 4

Iterate multiple slices with known upper bound

```
func bce(a, b, c []int) {
    for i := range a {
        a[i] = b[i] + c[i]
    }
}
```

```
$go tool compile -d=s/c/d=1 foo.go
foo.go:11:11: Found IsInBounds
foo.go:11:18: Found IsInBounds
```

Pattern 4

Iterate multiple slices with known upper bound

```
func bce(a, b, c []int) {
    _ = b[len(a)-1]
    _ = c[len(a)-1]
    for i := 0; i < len(a);
i++ {
        for i := range a {
            a[i] = b[i] + c[i]
        }
    }
}
```

```
$go tool compile -d=s/c/d=1 foo.go
foo.go:4:7: Found IsInBounds
foo.go:5:7: Found IsInBounds
```

Honorable mention

Slice range can be derived

```
func bce(b []byte, h []int32)
{
    for _, t := range b {
        h[t]++
    }
}
```

```
$go tool compile -d=s/c/d=1 foo.go
foo.go:5:4: Found IsInBounds
```

Honorable mention

Slice range can be derived

```
func bce(b []byte, h []int32)
{
    h = h[:256]
    for _, t := range b {
        h[t]++
    }
}
```

```
$go tool compile -d=s/c/d=1 foo.go
foo.go:4:7: Found IsSliceInBounds
```

SECTION FOUR

Demo exercise

DEMO

SECTION FIVE

Things to keep in mind

- Use it ONLY to squeeze the last drop of performance from your app. Sometimes shaving off a ms may or not may not matter.
- Always run benchmarks.
- Check against the latest Go version.

That's it!