The Go low-level calling convention on x86-64

Raphael 'kena' Poss

July 2018

modified: 2018-09-03
slug: go-calling-convention-x86-64
category: Programming
tags: golang, compilers, analysis, programming languages, c++
series: Go low-level code analysis

Contents

> **Note**
>
> The latest version of this document can be found online at https://dr-knz.net/go-calling-convention-x86-64.html. Alternate formats: Source, PDF.

Introduction

This article analyzes how the Go compiler generates code for function calls, argument passing and exception handling on x86-64 targets.

This expressely does not analyze how the Go compiler lays out data in memory (other than function arguments and return values), how escape analysis works, what the code generator must do to accommodate the asynchronous garbage collector, and how the handling of goroutines impacts code generation.

All tests below are performed using the `freebsd/amd64` target of go 1.10.3. The assembly listing are produced with `go tool objdump` and use the Go assembler syntax.

> **Note**
>
> A followup analysis is also available which revisits the findings with Go 1.15. Spoiler alert: only few things have changed. The 2018 findings remain largely valid.

Argument and return values, call sequence

Arguments and return value

How does Go pass arguments to function and return results?

Let us look at the simplest function:

```go
func EmptyFunc() { }
```

This compiles to:

```
EmptyFunc:
    0x480630            c3                  RET
```

Now, with a return value:

```
func FuncConst() int { return 123 }
```

This compiles to:

```
FuncConst:
    0x480630        48c74424087b000000    MOVQ $0x7b, 0x8(SP)
  0x480639        c3                    RET
```

So return values are passed via memory, on the stack, not in registers like in most standard x86-64 calling conventions for natively compiled languages.

Compare the output from a C or C++ compiler:

```
FuncConst:
    movl    $123, %eax
    retq
```

This passes the return value in a register.

How do simple arguments get passed in Go?

```
// Note: substracting z so we know which argument is which.
func FuncAdd(x,y,z int) int { return x + y - z }
```

This compiles to:

```
FuncAdd:
  0x480630        488b442408        MOVQ 0x8(SP), AX  ; get arg x
  0x480635        488b4c2410        MOVQ 0x10(SP), CX ; get arg y
  0x48063a        4801c8            ADDQ CX, AX       ; %ax <- x + y
  0x48063d        488b4c2418        MOVQ 0x18(SP), CX ; get arg z
  0x480642        4801c8            SUBQ CX, AX       ; %ax <- x + y - z
  0x480645        4889442420        MOVQ AX, 0x20(SP) ; return x+y-z
  0x48064a        c3                RET
```

So arguments are passed via memory, on the stack, not in registers like other languages.

Also we see the arguments are at the top of the stack, and the return value slot underneath that.

Compare the output from a C or C++ compiler:

```
FuncAdd:
    leal    (%rdi,%rsi), %eax
    subl    %edx, %eax
    retq
```

This passes the arguments in registers. The exact number depends on the calling convention, but for freebsd/amd64 up to 6 arguments are passed in registers, the rest on the stack.

Note that there is an open proposal to implement register passing in Go at https://github.com/golang/go/issues/18597. This proposal has not yet been accepted.

Call sequence: how a function gets called

How does a function like `FuncAdd` above get called?

```
func DoCallAdd() int { return FuncAdd(1, 2, 3) }
```

This gives:

```
0x480650    64488b0c25f8ffffff    MOVQ FS:0xfffffff8, CX
0x480659    483b6110              CMPQ 0x10(CX), SP
0x48065d    7641                  JBE 0x4806a0
0x48065f    4883ec28              SUBQ $0x28, SP
0x480663    48896c2420            MOVQ BP, 0x20(SP)
0x480668    488d6c2420            LEAQ 0x20(SP), BP
0x48066d    48c7042401000000      MOVQ $0x1, 0(SP)
0x480675    48c744240802000000    MOVQ $0x2, 0x8(SP)
0x48067e    48c744241003000000    MOVQ $0x3, 0x10(SP)
0x480687    e8a4ffffff            CALL src.FuncAdd(SB)
0x48068c    488b442418            MOVQ 0x18(SP), AX
0x480691    4889442430            MOVQ AX, 0x30(SP)
0x480696    488b6c2420            MOVQ 0x20(SP), BP
0x48069b    4883c428              ADDQ $0x28, SP
0x48069f    c3                    RET
0x4806a0    e80ba5fcff            CALL runtime.morestack_noctxt(SB)
0x4806a5    eba9                  JMP src.DoCallAdd(SB)
```

Woah, what is going on?
At the center of the function we see what we wanted to see:

```
0x48066d    48c7042401000000      MOVQ $0x1, 0(SP)     ; set arg x
0x480675    48c744240802000000    MOVQ $0x2, 0x8(SP)   ; set arg y
0x48067e    48c744241003000000    MOVQ $0x3, 0x10(SP)  ; set arg z
0x480687    e8a4ffffff            CALL src.FuncAdd(SB) ; call
0x48068c    488b442418            MOVQ 0x18(SP), AX    ; get return value of FuncAdd
0x480691    4889442430            MOVQ AX, 0x30(SP)    ; set return value of DoCallAdd
```

The arguments are pushed into the stack before the call, and after the call the return value is retrieved from the callee frame and copied to the caller frame. So far, so good.

However, now we know that arguments are passed on the stack, this means that any function that calls other functions now must ensure there is some stack space to pass arguments to its callees. This is what we see here:

```
; Before the call: make space for callee.
0x48065f    4883ec28              SUBQ $0x28, SP
; After the call: restore stack pointer.
0x48069b    4883c428              ADDQ $0x28, SP
```

Now, what is the remaining stuff?

Because Go has exceptions ("panics") it must preserve the ability of the runtime system to unwind the stack. So in every activation record it must store the difference between the stack pointer on entry and the stack pointer for callees. This is the "frame pointer" which is stored in this calling convention in the BP register. That is why we see:

```
; Store the frame pointer of the caller into a known location in
; the current activation record.
0x480663            48896c2420            MOVQ BP, 0x20(SP)
; Store the address of the copy of the parent frame pointer
; into the new frame pointer.
0x480668            488d6c2420            LEAQ 0x20(SP), BP
```

This maintains the invariant of the calling convention that BP always points to a linked list of frame pointers, where each successive value of BP is 32 bytes beyond the value of the stack pointer in the current frame (SP+0x20). This way the stack can always be successfully unwound.

Finally, what about the last bit of code?

```
0x480650            64488b0c25f8ffffff    MOVQ FS:0xfffffff8, CX
0x480659            483b6110              CMPQ 0x10(CX), SP
0x48065d            7641                  JBE 0x4806a0
...
0x4806a0            e80ba5fcff            CALL runtime.morestack_noctxt(SB)
0x4806a5            eba9                  JMP src.DoCallAdd(SB)
```

The Go runtime implements "tiny stacks" as an optimization: a goroutine always starts with a very small stack so that a running go program can have many "small" goroutines active at the same time. However that means that on the standard tiny stack it is not really possible to call many functions recursively.

Therefore, in Go, every function that needs an activation record on the stack needs first to check whether the current goroutine stack is large enough for this. It does this by comparing the current value of the stack pointer to the low water mark of the current goroutine, stored at offset 16 (0x10) of the goroutine struct, which itself can always be found at address FS:0xfffffff8.

Compare how `DoCallAdd` works in C or C++:

```
DoCallAdd:
        movl    $3, %edx
        movl    $2, %esi
        movl    $1, %edi
        jmp     FuncAdd
```

This passes the arguments in registers, then transfers control to the callee with a `jmp` — a tail call. This is valid because the return value of `FuncAdd` becomes the return value of `DoCallAdd`.

What of the stack pointer? The function `DoCallAdd` cannot tell us much in C because, in contrast to Go, it does not have any variables on the stack and thus does need an activation record. In general (and that is valid for Go too), if there is no need for an activation record, there is no need to set up / adjust the stack pointer.

So how would a C/C++ compiler handle an activation record? We can force one like this:

```
void other(int *x);
int DoCallAddX() { int x = 123; other(&x); return x; }
```

Gives us:

```
DoCallAddX:
       subq    $24, %rsp       ; make space
       leaq    12(%rsp), %rdi  ; allocate x at address rsp+12
       movl    $123, 12(%rsp)  ; store 123 into x
       call    other           ; call other(&x)
       movl    12(%rsp), %eax  ; load value from x
       addq    $24, %rsp       ; restore stack pointer
       ret
```

So `%rsp` gets adjusted upon function entry and restored in the epilogue.

No surprise. But is there? What of exception handling?

Aside: exceptions in C/C++

The assembly above was generated with a C/C++ compiler that *does* support exceptions. In general, the compiler cannot assume that a callee won't throw an exception. Yet we did not see anything about saving the stack pointer and/or setting up a frame pointer in the generated code above. So how does the C/C++ runtime handle stack unwinding?

There are fundamentally two main ways to implement exception propagation in an ABI (Application Binary Interface):

- "dynamic registration", with frame pointers in each activation record, organized as a linked list. This makes stack unwinding fast at the expense of having to set up the frame pointer in each function that calls other functions. This is also simpler to implement.
- "table-driven", where the compiler and assembler create data structures *alongside* the program code to indicate which addresses of code correspond to which sizes of activation records. This is called "Call Frame Information" (CFI) data in e.g. the GNU tool chain. When an exception is generated, the data in this table is loaded to determine how to unwind. This makes exception propagation slower but the general case faster.

In general, a language where exceptions are common and used for control flow will adopt dynamic registration, whereas a language where exceptions are rare will adopt table-driven unwinding to ensure the common case is more efficient. The latter choice is extremely common for C/C++ compilers.

Interestingly, the Go language designers recommend *against* using exceptions ("panics") for control flow, so one would expect they expect their language to fall in the second category and ought to also implement table-driven unwinding. Yet the Go compiler still uses dynamic registration. Maybe the table-driven approach was not used because it is more complex to implement?

More reading:

- JL Schilling - Optimizing away C++ exception handling - ACM SIGPLAN Notices, 1998

Callee-save registers—or not

Are there callee-save registers in Go? Can the Go compiler expect the callee will avoid using some registers, i.e. they won't be clobbered unless strictly needed?

In other languages, this optimization enables a function that calls another function to keep "important" values in registers and avoid to push its temporary variables to the stack (and thus force the apparition of an activation record on the stack).

Let's try:

```
func Intermediate() int {
  x := Other()
  x += Other()
  return x
}
```

Is there a callee-save register for the Go compiler to store `x` in?

Let's check:

```
Intermediate:
  [...]
  0x4806dd            e8ceffffff            CALL src.Other(SB)
  0x4806e2            488b0424              MOVQ 0(SP), AX
  0x4806e6            4889442408            MOVQ AX, 0x8(SP)
  0x4806eb            e8c0ffffff            CALL src.Other(SB)
  0x4806f0            488b442408            MOVQ 0x8(SP), AX
  0x4806f5            48030424              ADDQ 0(SP), AX
  0x4806f9            4889442420            MOVQ AX, 0x20(SP)
  [...]
```

So, no. The Go compiler always spills the temporaries to the stack during calls.

What does the C/C++ compiler do for this? Let's see:

```
Intermediate:
    pushq   %rbx          ; save %rbx from caller
    xorl    %eax, %eax
    call    other
    movl    %eax, %ebx    ; use callee-save for intermediate result
    xorl    %eax, %eax
    call    other
    addl    %ebx, %eax    ; use callee-save again
    popq    %rbx          ; restore callee-save for caller
    ret
```

Most C/C++ calling convention have a number of callee-save registers for intermediate results. On this platform, this includes at least `%rbx`.

The cost of pointers and interfaces

Go implements both pointer types (e.g. `*int`) and interface types with vtables (comparable to classes containing virtual methods in C++).

How are they implemented in the calling convention?

Pointers use just one word

Looking at the following code:

```
func UsePtr(x *int) int { return *x }
```

The generated code:

```
UsePtr:
  0x480630            488b442408            MOVQ 0x8(SP), AX  ; load x
  0x480635            488b00                MOVQ 0(AX), AX    ; load *x
  0x480638            4889442410            MOVQ AX, 0x10(SP) ; return *x
  0x48063d            c3                    RET
```

So a pointer is the same size as an `int` and uses just one word slot in the argument struct. Ditto for return values:

```
var x int
func RetPtr() *int { return &x }
func NilPtr() *int { return nil }
```

This gives us:

```
RetPtr:
  0x480650          488d0581010c00        LEAQ src.x(SB), AX  ; compute &x
  0x480657          4889442408            MOVQ AX, 0x8(SP)    ; return &x
  0x48065c          c3                    RET
NilPtr:
  0x480660          48c744240800000000    MOVQ $0x0, 0x8(SP)  ; return 0
  0x480669          c3                    RET
```

Interfaces use two words

Considering the following code:

```
type Foo interface{ foo() }
func InterfaceNil() Foo { return nil }
```

The compiler generates the following:

```
InterfaceNil:
  0x4805b0          0f57c0                XORPS X0, X0
  0x4805b3          0f11442408            MOVUPS X0, 0x8(SP)
  0x4805b8          c3                    RET
```

So an interface value is bigger. The pseudo-register `X0` in the Go pseudo-assembly is really the x86 `%xmm0`, a full 16-byte (128 bit) register.

We can confirm that by looking at a function that simply forwards an interface argument as a return value:

```
func InterfacePass(Foo x) Foo { return x }
```

This gives us:

```
InterfacePass:
  0x4805b0          488b442408            MOVQ 0x8(SP), AX
  0x4805b5          4889442418            MOVQ AX, 0x18(SP)
  0x4805ba          488b442410            MOVQ 0x10(SP), AX
  0x4805bf          4889442420            MOVQ AX, 0x20(SP)
  0x4805c4          c3                    RET
```

Although there is just 1 argument and return value, the compiler has to copy two words. Interface "values" are really a pointer to a vtable and a value combined together.

Strings and slices use two and three words

Next to pointers (one word) and interface values (two words) the Go compiler also has special layouts for two other things:

- the special type `string` is implemented as two words: a word containing the length, and a word to the start of the string. This supports computing `len()` and slicing in constant time.
- all slice types (`[]T`) are implemented using 3 words: a length, a capacity and a pointer to the first element. This supports computing `len()`, `cap()` and slicing in constant time.

The reason why `string` values do not need a capacity is that `string` is an immutable type in Go.

Constructing interface values

> **Note**
>
> The information in this section was collected with Go 1.10. It remains largely unchanged in Go 1.15. The minor differences are reviewed in this followup analysis.

Constructing a non-nil interface value requires storing the vtable pointer alongside the value.

In most real world cases the vtable part is known statically (because the type being cast to the interface type is known statically). We'll ignore the conversions from one interface type to another here.

For the value part, Go has multiple implementation strategies based on the actual type of value.

The most common case, an interface implemented by a pointer type, looks like this:

```
// Define the interface.
type Foo interface{ foo() }
// Define a struct type implementing the interface by pointer.
type foo struct{ x int }
func (*foo) foo() {}
// Define a global variable so we don't use the heap allocator.
var x foo

// Make an interface value.
func MakeInterface1() Foo { return &x }
```

This gives us:

```
MakeInterface1:
    0x4805c0          488d05d9010400        LEAQ go.itab.*src.foo,src.Foo(SB), AX
    0x4805c7          4889442408            MOVQ AX, 0x8(SP)
    0x4805cc          488d0505f20b00        LEAQ src.x(SB), AX
    0x4805d3          4889442410            MOVQ AX, 0x10(SP)
    0x4805d8          c3                    RET
```

Just as predicted: address of vtable in the first word, pointer to the struct in the second word. No surprise.

Things become a bit more expensive if the struct implements the interface by value:

```
// Define a struct type implementing the interface by value.
type bar struct{ x int }
func (bar) foo() {}
// Define a global variable so we don't use the heap allocator.
var y bar

// Make an interface value.
func MakeInterface2() Foo { return y }
```

This gives us:

```
MakeInterface2:
    0x4805c0          64488b0c25f8ffffff    MOVQ FS:0xfffffff8, CX
    0x4805c9          483b6110              CMPQ 0x10(CX), SP
    0x4805cd          7648                  JBE 0x480617
    0x4805cf          4883ec28              SUBQ $0x28, SP
    0x4805d3          48896c2420            MOVQ BP, 0x20(SP)
    0x4805d8          488d6c2420            LEAQ 0x20(SP), BP
    0x4805dd          488d053c020400        LEAQ go.itab.src.bar,src.Foo(SB), AX
```

```
0x4805e4          48890424          MOVQ AX, 0(SP)
0x4805e8          488d05e9f10b00    LEAQ src.x(SB), AX
0x4805ef          4889442408        MOVQ AX, 0x8(SP)
0x4805f4          e8e7b2f8ff        CALL runtime.convT2I64(SB)
0x4805f9          488b442410        MOVQ 0x10(SP), AX
0x4805fe          488b4c2418        MOVQ 0x18(SP), CX
0x480603          4889442430        MOVQ AX, 0x30(SP)
0x480608          48894c2438        MOVQ CX, 0x38(SP)
0x48060d          488b6c2420        MOVQ 0x20(SP), BP
0x480612          4883c428          ADDQ $0x28, SP
0x480616          c3                RET
0x480617          e814a5fcff        CALL runtime.morestack_noctxt(SB)
0x48061c          eba2              JMP github.com/knz/go-panic/src.MakeInterface2(SB)
```

Holy Moly. What just went on?

The function became suddently much larger because it is now making a call to another function `runtime.convT2I64`.

As per the previous sections, as soon as there is a callee, the caller must set up an activation record, so we see 1) a check the stack is large enough 2) adjusting the stack pointer 3) preserving the frame pointer for stack unwinding during exceptions. This explains the prologue and epilogue, so the "meat" that remains, taking this into account, is this:

```
0x4805dd          488d053c020400    LEAQ go.itab.src.bar,src.Foo(SB), AX
0x4805e4          48890424          MOVQ AX, 0(SP)
0x4805e8          488d05e9f10b00    LEAQ src.x(SB), AX
0x4805ef          4889442408        MOVQ AX, 0x8(SP)
0x4805f4          e8e7b2f8ff        CALL runtime.convT2I64(SB)
0x4805f9          488b442410        MOVQ 0x10(SP), AX
0x4805fe          488b4c2418        MOVQ 0x18(SP), CX
0x480603          4889442430        MOVQ AX, 0x30(SP)
0x480608          48894c2438        MOVQ CX, 0x38(SP)
```

What this does is to perform the regular Go call `runtime.convT2I64(&bar_foo_vtable, y)` and returns its result, which is an interface and thus takes two words.

What does this function do?

```go
func convT2I64(tab *itab, elem unsafe.Pointer) (i iface) {
    t := tab._type
    // [...]
    var x unsafe.Pointer
    // [...]
            x = mallocgc(8, t, false)
            *(*uint64)(x) = *(*uint64)(elem)
    // [...]
    i.tab = tab
    i.data = x
    return
}
```

What this does really is to call the heap allocator and allocate a slot in memory to store a copy of the value provided, and a pointer to that heap-allocated slot is stored in the interface value.

In other words, in general, types that implement interfaces by value will mandate a trip to the heap allocator every time a value of that type is turned into an interface value.

As a special case, if the value provided is the "zero value" for the type implementing the interface, the heap allocation is avoided and a special "address to the zero value" is used instead to construct the interface reference. This is checked by `convT2I64` in the code I elided above:

```
if *(*uint64)(elem) == 0 {
        x = unsafe.Pointer(&zeroVal[0])
} else {
        x = mallocgc(8, t, false)
        *(*uint64)(x) = *(*uint64)(elem)
}
```

This is correct because the function `convT2I64` is only used for 64-bit types that implement the interface. This is true of the `struct` that I defined above, which contains just one 64-bit field.

There are many such `convT2I` functions for various type layouts that may implement the interface, for example:

- `convT2I16`, `convT2I32`, `convT2I64` for "small" types;
- `convT2Istring`, `convT2Islice` for `string` and slice types;
- `convT2Inoptr` for structs that do not contain pointers;
- `convT2I` for the general case.

All of them except for the general cases `convT2Inoptr` and `convT2I` will attempt to avoid the heap allocator if the value is the zero value.

Nevertheless, in all these cases the caller that is constructing an interface value must check its stack size and set up an activation record, because it is making a call.

So, in general, types that implement interfaces by value cause overhead when they are converted into the interface type.

Interfaces for empty structs

There is just one, not-too-exciting super-special case: *empty structs*. These can implement the interface by value without overhead:

```
type empty struct{}
func (empty) foo() {}
var x empty
func MakeInterface3() Foo { return x }
```

This gives us:

```
MakeInterface3:
  0x4805c0          488d0539020400          LEAQ go.itab.src.empty,src.Foo(SB), AX
  0x4805c7          4889442408              MOVQ AX, 0x8(SP)
  0x4805cc          488d05edf20b00          LEAQ runtime.zerobase(SB), AX
  0x4805d3          4889442410              MOVQ AX, 0x10(SP)
  0x4805d8          c3                      RET
```

The "value part" of interface values for empty structs is always `&runtime.zerobase` and can be computed without a call and thus without overhead.

`error` is an interface type

Compare the following two functions:

```
func Simple1() int { return 123 }
func Simple2() (int, error) { return 123, nil }
```

And their generated code:

```
Simple1:
  0x4805b0            48c74424087b000000      MOVQ $0x7b, 0x8(SP)
  0x4805b9            c3                      RET

Simple2:
  0x4805c0            48c74424087b000000      MOVQ $0x7b, 0x8(SP)
  0x4805c9            0f57c0                  XORPS X0, X0
  0x4805cc            0f11442410              MOVUPS X0, 0x10(SP)
  0x4805d1            c3                      RET
```

What we see here is that `error` being an interface, the function returning `error` must set up two extra words of return value.

In the `nil` case this is still straightforward (at the expense of 16 bytes of extra zero data).

It is also still pretty straightforward if the error object was pre-allocated. For example:

```
var errDivByZero = errors.New("can't divide by zero")

func Compute(x, y float64) (float64, error) {
  if y == 0 {
    return 0, errDivByZero
  }
  return x / y, nil
}
```

Compiling to:

```
Compute:
  0x4805e0            f20f10442410            MOVSD_XMM 0x10(SP), X0 ; load y into X0
        if y == 0 {
  0x4805e6            0f57c9                  XORPS X1, X1    ; compute float64(0)
  0x4805e9            660f2ec1                UCOMISD X1, X0  ; is y == 0?
  0x4805ed            7521                    JNE 0x480610    ; no: go to return x/y
  0x4805ef            7a1f                    JP 0x480610     ; no: go to return x/y
              return 0, errDivByZero
  0x4805f1            488b05402c0a00          MOVQ src.errDivByZero+8(SB), AX
  0x4805f8            488b0d312c0a00          MOVQ src.errDivByZero(SB), CX
  0x4805ff            f20f114c2418            MOVSD_XMM X1, 0x18(SP)
  0x480605            48894c2420              MOVQ CX, 0x20(SP)
  0x48060a            4889442428              MOVQ AX, 0x28(SP)
  0x48060f            c3                      RET
        return x / y, nil
  0x480610            f20f104c2408            MOVSD_XMM 0x8(SP), X1   ; load x into X1
  0x480616            f20f5ec8                DIVSD X0, X1            ; compute x / y
  0x48061a            f20f114c2418            MOVSD_XMM X1, 0x18(SP)  ; return x / y
  0x480620            0f57c0                  XORPS X0, X0            ; compute error(nil)
  0x480623            0f11442420              MOVUPS X0, 0x20(SP)     ; return error(nil)
  0x480628            c3                      RET
```

Common case: computed errors

The simple case where error objects are pre-allocated is handled efficiently, but in real world code the error text is usually computed to include some contextual information, for example:

```
func Compute(x, y float64) (float64, error) {
  if y == 0 {
    return 0, fmt.Errorf("can't divide %f by zero", x)
  }
  return x / y, nil
}
```

At the moment we organize the function this way, we are paying the price of a call to another function: setting up an activation record, frame pointer, checking the stack size, etc. Even on the "hot" path where the error does not occur.

This makes the relatively "simple" function `Compute`, where the crux of the computation is just 1 instruction, `divsd`, extremely large:

```
Compute:
  ; [... stack size check, SP and BP set up elided ...]
  0x482201            f20f10442468            MOVSD_XMM 0x68(SP), X0  ; load y
        if y == 0 {    ; like before
  0x482207            0f57c9                  XORPS X1, X1   ; compute float64(0)
  0x48220a            660f2ec1                UCOMISD X1, X0 ; is y == 0?
  0x48220e            0f85a7000000            JNE 0x4822bb   ; no: go to return x/y
  0x482214            0f8aa1000000            JP 0x4822bb    ; no: go to return x/y

                return 0, fmt.Errorf("can't divide %f by zero", x)
  0x48221a            f20f10442460            MOVSD_XMM 0x60(SP), X0  ; load x

  ; The following code allocates a special struct using
  ; runtime.convT2E64 to pass the variable arguments to
  ; fmt.Errorf. The struct contains the value of x.
  0x482220            f20f11442438            MOVSD_XMM X0, 0x38(SP)
  0x482226            0f57c0                  XORPS X0, X0
  0x482229            0f11442440              MOVUPS X0, 0x40(SP)
  0x48222e            488d05cbf80000          LEAQ 0xf8cb(IP), AX
  0x482235            48890424                MOVQ AX, 0(SP)
  0x482239            488d442438              LEAQ 0x38(SP), AX
  0x48223e            4889442408              MOVQ AX, 0x8(SP)
  0x482243            e83895f8ff              CALL runtime.convT2E64(SB)
  0x482248            488b442410              MOVQ 0x10(SP), AX
  0x48224d            488b4c2418              MOVQ 0x18(SP), CX

  ; The varargs struct is saved for later on the stack.
  0x482252            4889442440              MOVQ AX, 0x40(SP)
  0x482257            48894c2448              MOVQ CX, 0x48(SP)

  ; The constant string "can't divide..." is passed in the argument list of fmt.Errorf.
  0x48225c            488d05111e0300          LEAQ 0x31e11(IP), AX
  0x482263            48890424                MOVQ AX, 0(SP)
  0x482267            48c744240817000000      MOVQ $0x17, 0x8(SP)

  ; A slice object is created to point to the vararg struct and given
  ; as argument to fmt.Errorf.
  0x482270            488d442440              LEAQ 0x40(SP), AX
  0x482275            4889442410              MOVQ AX, 0x10(SP)
```

```
0x48227a              48c744241801000000      MOVQ $0x1, 0x18(SP)
0x482283              48c744242001000000      MOVQ $0x1, 0x20(SP)
0x48228c              e8cf81ffff              CALL fmt.Errorf(SB)
; The result value of fmt.Errorf is retrieved.
0x482291              488b442428              MOVQ 0x28(SP), AX
0x482296              488b4c2430              MOVQ 0x30(SP), CX

; return float64(0) as first return value:
0x48229b              0f57c0                  XORPS X0, X0
0x48229e              f20f11442470            MOVSD_XMM X0, 0x70(SP)
; return the result of fmt.Errorf as 2nd return value:
0x4822a4              4889442478              MOVQ AX, 0x78(SP)
0x4822a9              48898c2480000000        MOVQ CX, 0x80(SP)
; [ ... restore BP/SP ... ]
0x4822ba              c3                      RET

      return x / y, nil    ; same as before
0x4822bb              f20f104c2460            MOVSD_XMM 0x60(SP), X1 ; load x into X1
0x4822c1              f20f5ec8                DIVSD X0, X1           ; compute x / y
0x4822c5              f20f114c2470            MOVSD_XMM X1, 0x70(SP) ; return x / y
0x4822cb              0f57c0                  XORPS X0, X0           ; compute error(nil)
0x4822ce              0f11442478              MOVUPS X0, 0x78(SP)    ; return error(nil)
; [ ... restore BP/SP ... ]
0x4822dc              c3                      RET
```

So what are we learning here?

- `fmt.Errorf` (like all vararg functions in Go) get additional argument passing code: the arguments are stored in the caller's activation record, and a slice object is given as argument to the vararg-accepting callee.
- this price is paid on the cold path to any real-world function that allocates error objects dynamically when an error is encountered.

We are not considering here the cost of running `fmt.Errorf` itself, which usually has to go to the heap allocator multiple times because it does not know in advance how long the computed string will be.

> **Note**
> A more thorough review of how vararg calls work is available in this followup analysis.

Common case: testing for errors

The other common case is when a caller checks the error returned by a callee, like this:

```go
func Caller() (int, error) {
  v, err := Callee()
  if err != nil {
    return -1, err
  }
  return v + 1, nil
}
```

This gives us:

```
Caller:
  ; [... stack size check, SP and BP set up elided ...]
        v, err := Callee()
  0x48061d              e8beffffff              CALL src.Callee(SB)
  0x480622              488b442410              MOVQ 0x10(SP), AX  ; retrieve return value
  0x480627              488b0c24                MOVQ 0(SP), CX     ; load error vtable
  0x48062b              488b542408              MOVQ 0x8(SP), DX   ; load error value
        if err != nil {
  0x480630              4885d2                  TESTQ DX, DX       ; is the value part nil?
  0x480633              741d                    JE 0x480652        ; yes, go to v+1 below
                return -1, err
  0x480635              48c7442428ffffffff      MOVQ $-0x1, 0x28(SP)  ; return -1
  0x48063e              4889542430              MOVQ DX, 0x30(SP)     ; return err.vtable
  0x480643              4889442438              MOVQ AX, 0x38(SP)     ; return err.value
  ; [ ... restore BP/SP ... ]
  0x480651              c3                      RET
        return v + 1, nil
  0x480652              488d4101                LEAQ 0x1(CX), AX    ; compute v + 1
  0x480656              4889442428              MOVQ AX, 0x28(SP)   ; return v + 1
  0x48065b              0f57c0                  XORPS X0, X0        ; compute error(nil)
  0x48065e              0f11442430              MOVUPS X0, 0x30(SP) ; return error(nil)
  ; [ ... restore BP/SP ... ]
  0x48066c              c3                      RET
```

So any time a caller needs to check the error return of a callee, there are 2 instructions to retrieve the error value, 2 instructions to test whether it is `nil`, and in the "hot" path where there is no error two more instruction on every return path to return `error(nil)`.

For reference (we'll consider that again below), if there was no error to check/propagate the function becomes much simpler:

```
Caller:
   ; [... stack size check, SP and BP set up elided ...]
   0x48060d              e8ceffffff              CALL github.com/knz/go-panic/src.Callee2(SB)
   0x480612              488b0424                MOVQ 0(SP), AX     ; retrieve return value
   0x480616              48ffc0                  INCQ AX            ; compute v + 1
   0x480619              4889442418              MOVQ AX, 0x18(SP) ; return v + 1
   ; [ ... restore BP/SP ... ]
   0x480627              c3                      RET
```

(No extra instructions, no extra branch.)

Implementation of `defer`

> **Note**
>
> The mechanism presented in this section is still used as of Go 1.15. However, Go 1.15 has an optimization that is enabled in a common case which simplifies the mechanism further. We will see how that works in this followup analysis.

Go provides a feature to register, from the body of a function, a list of callback functions that are *guaranteed* to be called when the call terminates, even during exception propagation.

(This is useful e.g. to ensure that resources are freed and mutexes unlocked regardless of what happens with one of the callees.)

How does this work? Let's consider the simple example:

```
func Defer1() int { defer f(); return 123 }
```

This compiles to:

```
Defer1:
    ; [... stack size check, SP and BP set up elided ...]

    ; Prepare the return value 0. This is set in memory because
    ; (theoretically, albeit not in this particular example) the deferred
    ; function can access the return value and may do so before it was
    ; set by the remainder of the function body.
    0x48208d            48c744242000000000      MOVQ $0x0, 0x20(SP)

    ; Prepare the defer by calling runtime.deferproc(0, &f)
    0x482096            c7042400000000          MOVL $0x0, 0(SP)
    0x48209d            488d05f46e0300          LEAQ 0x36ef4(IP), AX
    0x4820a4            4889442408              MOVQ AX, 0x8(SP)
    0x4820a9            e8822afaff              CALL runtime.deferproc(SB)

    ; Special check of the return value of runtime.deferproc.
    ; In the common case, deferproc returns 0.
    ; If a panic is generated by the function body (or one of the callees),
    ; and the defer function catches the panic with `recover`, then
    ; control will re-return from `deferproc` with value 1.
    0x4820ae            85c0                    TESTL AX, AX
    0x4820b0            7519                    JNE 0x4820cb  ; has a panic been caught?

    ; Prepare the return value 123.
    0x4820b2            48c74424207b000000      MOVQ $0x7b, 0x20(SP)
    0x4820bb            90                      NOPL
    ; Ensure the defers are run.
    0x4820bc            e84f33faff              CALL runtime.deferreturn(SB)
    ; [ ... restore BP/SP ... ]
    0x4820ca            c3                      RET

    ; We've caught a panic. We're still running the defers.
    0x4820cb            90                      NOPL
    0x4820cc            e83f33faff              CALL runtime.deferreturn(SB)
    ; [ ... restore BP/SP ... ]
    0x4820da            c3                      RET
```

How to read this:

- the code generated for function that contains `defer` always contains calls to `deferproc` and `defereturn` and thus needs an activation record, and thus a stack size check and frame pointer setup.
- if a function contains `defer` there will be a call to `deferreturn` on every return path.
- the actual callback is not stored in the activation record of the function; instead what `deferproc` does (internally) is store the callback in a linked list from the goroutine's header struct. `deferreturn` runs and pops the entries from that linked list.

The code is generated this way regardless of whether the deferred function contains `recover()`, see below.

Deferred closures

In real-world uses, the deferred function is actually a closure that has access to the enclosing function's local variables. For example:

```
func Defer2() (res int) {
    defer func() {
            res = 123
    }()
    return -1
}
```

This compiles to:

```
 Defer2:
    ; [... stack size check, SP and BP set up elided ...]

    ; Store the zero value as return value.
    0x48208d              48c744242800000000      MOVQ $0x0, 0x28(SP)

    ; Store the frame pointer of Defer2 for use by the deferred closure.
    0x482096              488d442428              LEAQ 0x28(SP), AX
    0x48209b              4889442410              MOVQ AX, 0x10(SP)

    ; Call runtime.deferproc(8, &Defer2.func1)
    ; Where Defer2.func1 is the code generated for the closure, see below.
    ; The closure takes an implicit argument, which is the frame
    ; pointer of the enclosing function, where it can peek
    ; at the enclosing function's local variables.
    0x4820a0              c7042408000000          MOVL $0x8, 0(SP)
    0x4820a7              488d05b26e0300          LEAQ 0x36eb2(IP), AX
    0x4820ae              4889442408              MOVQ AX, 0x8(SP)
    0x4820b3              e8782afaff              CALL runtime.deferproc(SB)

    ; Are we recovering from a panic?
    0x4820b8              85c0                    TESTL AX, AX
    0x4820ba              7519                    JNE 0x4820d5

    ; Common path.
    ; Set -1 as return value.
    0x4820bc              48c744242800000000      MOVQ $-1, 0x28(SP)
    0x4820c5              90                      NOPL
    ; Run the defers.
    0x4820c6              e84533faff              CALL runtime.deferreturn(SB)
    ; [ ... restore BP/SP ... ]
    0x4820d4              c3                      RET

    ; Recovering from a panic.
    0x4820d5              90                      NOPL
    0x4820d6              e83533faff              CALL runtime.deferreturn(SB)
    ; [ ... restore BP/SP ... ]
    0x4820e4              c3                      RET

Defer2.func1:
    ; Load the frame pointer of the enclosing function.
    MOVQ    0x8(SP), AX
    ; Store the new value into the return value slot of the
    ; enclosing function's frame.
    MOVQ    $123, (AX)
    RET
```

So a closure gets compiled as an anonymous function which returns a pointer to the enclosing frame as implicit first argument.

Every non-local variable accessed in the closure is marked to force spill in the enclosing function, to ensure they are allocated on the stack and not in registers.

Since return values and arguments are always on the stack anyway, using them in closures thus comes at no additional overhead. This would be different for other variables which could avoid a stack allocation otherwise.

> **Note**
>
> This section focuses specifically on *deferred* closures. This gives the Go compiler the guarantee that the closure itsself *does not escape*.
> If the closure did escape, then additional machinery would kick in to allocate the closure on the heap together with the variables it needs to access from the enclosing function.

Implementation of `panic`

Using `panic()` in a function

A function that uses `panic()` without computing anything (including, for now, not computing any object as exception) looks like this:

```
func Panic1() { panic(nil) }
var x int
func Panic2() { panic(&x) }
```

This gives us:

```
Panic1:
  ; [... stack size check, SP and BP set up elided ...]
  0x4805fd          0f57c0              XORPS X0, X0
  0x480600          0f110424            MOVUPS X0, 0(SP)
  0x480604          e8f747faff          CALL runtime.gopanic(SB)
  0x480609          0f0b                UD2

Panic2:
  ; [... stack size check, SP and BP set up elided ...]
  0x4806dd          488d05bcaf0000      LEAQ 0xafbc(IP), AX
  0x4806e4          48890424            MOVQ AX, 0(SP)
  0x4806e8          488d05f1f00b00      LEAQ src.x(SB), AX
  0x4806ef          4889442408          MOVQ AX, 0x8(SP)
  0x4806f4          e80747faff          CALL runtime.gopanic(SB)
  0x4806f9          0f0b                UD2
```

What is going on?

Using `panic()` in the body of a function translates in any case to a call to `runtime.gopanic()`. Therefore in any case the function needs to check its stack size and set up an activation record, like every other function that calls anything.

Then for the call to `runtime.gopanic()`: this function takes a single argument of type `interface{}`. So the caller that invokes `panic()` must create an interface value with whatever object/value it wants to use as exception.

- In `Panic2()` the regular vtable for `interface{}` is used and the address of `x` is passed as interface value.
- In `Panic1()` the Go compiler uses another special-case optimization: `interface{}(nil)` is implemented using a zero vtable.

So really, from the perspective of generated code, using `panic()` in the body of a function looks very much like any other function call, except it is actually simpler: the compiler knows that `runtime.gopanic()` does not return and thus does not need to generate instructions to return the caller on the return path from the call to `gopanic`.

Finally, if the function needs to create/allocate an object to throw as exception, the code to prepare this object (initialization, allocation, etc.) will be added just as usual.

Exceptions for intermediate functions

The Go code generation of a function that calls another function that *may* throw an exception does not handle anything specially: it sets up an activation record and prepares the frame pointer as usual.

This price paid for setting up the frame pointer is paid anytime another function is called, irrespective of whether it will throw an exception or not.

Therefore, exception propagation in Go is cheaper than the testing and propagation of `error` results.

Catching exceptions: `defer` + `recover`

As of Go 1.10 the language does not provide a simple-to-use control structure like `try-catch`.

Instead, it provides a special pseudo-function called `recover()`. When the author of a function `foo()` wishes to catch an exception generated in `foo()` or one of its callees, the code must be structured as follows:

- a separate function or closure (other than `foo`) must contain a call to `recover()`;
- a call to that separate function must be deferred from `foo`.

Low-level mechanism

We can look at the mechanism by compiling the following:

```
func Recovering(r *int) {
    // The pseudo-function recover() returns nil by default, except when
    // called in a deferred activation, in which case it catches the
    // exception object, stops stack unwinding and returns the exception
    // object as its return value.
    if recover() != nil {
        *r = 123
    }
}

func TryCatch() (res int) {
    defer Recovering(&res)
    // call a function that may throw an exception.
    f()
    // Regular path: return -1
    res = -1
}
```

In this example, the `TryCatch` function is compiled like the functions `Defer1`/`Defer2` of the previous section, so it is not detailed further. The interesting part is `Recovering`:

```
Recovering:
   ; [... stack size check, SP and BP set up elided ...]
```

```
; Call runtime.gorecover(), giving it the address of Recover's
; activation record as argument.
0x48208d           488d442428           LEAQ 0x28(SP), AX
0x482092           48890424             MOVQ AX, 0(SP)
0x482096           e8653efaff           CALL runtime.gorecover(SB)

; Check the return value.
0x48209b           488b442408           MOVQ 0x8(SP), AX
0x4820a0           4885c0               TESTQ AX, AX  ; is it nil?
0x4820a3           740c                 JE 0x4820b1   ; yes, go to the return path below.

; Retrieve the argument r
0x4820a5           488b442428           MOVQ 0x28(SP), AX
; Set *r = 123
0x4820aa           48c7007b000000       MOVQ $0x7b, 0(AX)

0x4820b1
    ; [ ... restore BP/SP ... ]
0x4820ba           c3                   RET
```

Because using the pseudo-function `recover()` compiles to a function call, the `Recovering` function needs its own activation record, thus stack size check, frame pointer, etc.

What the `gorecover()` function internally does, in turn, is to check if there is an exception propagation in progress. If there is, it stops the propagation and returns the panic object. If there is not, it simply returns `nil`.

(To "stop the propagation" it sets a flag in the panic object / goroutine struct. This is subsequently picked up by the unwind mechanism when the deferred function terminates. See the source code in `src/runtime/panic.go` for details.)

Cost of `defer` + `recover`

A function that wishes to catch an exception needs to `defer` the other function that will actually do the catch.

This incurs the cost of `defer` always, even when the exception does not occur:

- setting up an activation record (checking the stack size, adjusting the stack pointer, setting the frame pointer, etc.) because there will be a call in any case.
- setting up the deferred call in the goroutine header struct at the beginning.
- performing the deferred call on every return path.

The first cost is only overhead if the function catching the exception did not otherwise contain function calls and could have avoided allocating an activation record. For example, a "small" function that merely accesses some existing structs and may only panic due to e.g. a nil pointer dereference, would see that cost as overhead.

The other two costs are relatively low:

- setting up the deferred call does not need heap allocation. Its code path is relatively short. The main price it pays is accessing the goroutine header struct and a couple conditional branches.
- running the deferred calls however incurs:

  - the price of jumping around the deferred call list (a few memory accesses but no conditional branch, so fairly innocuous);
  - running the body of the actual deferred functions. This in turn costs the overhead of setting up and tearing down *their* activation record (because they're probably calling other functions, e.g. when they use `recover`) even when there is no exception to catch.

We will look at empirical measurements in a separate article.

An interesting question: `error` vs `panic`?

What is cheaper: handling exceptions via `panic` / `recover`, or passing and testing error results with `if err := ...; err != nil { return err }`?

The analysis above so far reveals:

- at the point an exception/error is generated:

  - in both cases the function that generates an exception/error usually needs an activation record (stack size check, etc.)
    - for `panic`, because of the call to `runtime.gopanic`;
    - for both errors and panics, because of the call to `fmt.Errorf` or `fmt.Sprintf` to create a contextual error object.
  - in the specific case of a function that does not call other functions, and only returns pre-defined error objects, using `panic` will incur an activation record whereas the function with an error return will not need it.
  - throwing an exception with `panic` results in less code usually, because the compiler does not generate a return path.

  To summarise, overall, the two approaches for the function(s) where exceptions/errors occur have similar costs.

- in "leaf" functions that never produce exceptions/errors but must implement an interface type where other implementors of the interface *may* produce exceptions/errors, handling exceptions/errors with `panic` is *always cheaper.*

  This is because the leaf function will neither contain `panic` nor the initialization of the extra `nil` return value.

- at the point an exception/error is propagated without change, the `panic`-based handling is *always cheaper*:

  - it moves fewer result values around from callee to caller;
  - it does not contain a test of the error return and the accompanying conditional branch;
  - there is fewer code overall so less pressure on the I-cache.

- at the point an exception/error is caught and conditionally handled, then the `panic`-based handling is *always more expensive* because it incurs the cost of `defer` and an extra activation record (for the deferred closure/function) which the error-based handling does not require.

So in short, this is not a clear-cut case: `panic`-based exception handling is nearly-always cheaper for the tree of callees, but more expensive for the code that catches the exception.

Using `panic` over error returns is thus only advantageous if there is enough computation in the call tree to offset the cost of setting up the catch environment. This is true in particular:

- when the call tree where errors can be generated is guaranteed to always be deep/complex enough that the savings of the `panic`-based handling will be noticeable.
- when the call tree is invoked multiple times and the catch environment can be set up just once for all the calls.

I aim to complement this article with a later experiment to verify this hypothesis empirically.

Differences with `gccgo`

> **Important**
>
> (Erratum as of January 2020): The following observations were made in 2018 with all optimizations disabled. This is unfair to `gccgo`, as GCC's optimization engine is quite capable. Once enabled with `-O`, `gccgo` is in fact able to use registers to pass arguments. I might revisit this topic in a followup article..

The GNU Compiler Collection now contains a Go compiler too called `gccgo`.

In contrast to `6g` (the original Go compiler) this tries to mimic the native calling convention. This brings *potential* performance benefits:

- arguments are passed in registers when possible.
- the first return value is passed in a register.
- attempts to use callee-save registers for temporaries.

However these benefits are not actually realized, because `gccgo` (as of GCC 8.2) also has the following problems:

- it disables many standard GCC optimizations, like register reloading and (some forms of) temporary variable elimination, and thus causes many more spills to memory than necessary.
- because the temporary variables spill to the stack nearly always, that means nearly every function needs an activation record (not just those that call other functions or have many local variables) and thus always need to check the stack size upfront.

These two limitations together make the code generated by `gccgo` unacceptably longer and more memory-heavy overall.

For example, the simple `FuncAdd` from the beginning of this document compiles with `gccgo` to:

```
FuncAdd:
  3b91:       64 48 3b 24 25 70 00    cmp    %fs:0x70,%rsp          ; is the stack large enough?
  3b98:       00 00
  3b9a:       73 12                     jae    3bae <src.FuncAdd+0x1d> ; yes, go below
  3b9c:       41 ba 08 00 00 00       mov    $0x8,%r10d             ; call __morestack
  3ba2:       41 bb 00 00 00 00       mov    $0x0,%r11d
  3ba8:       e8 36 10 00 00          callq  4be3 <__morestack>

  ; The following `retq` instruction on the return path to
  ; __morestack is not actually executed: `__morestack` is a standard
  ; GCC facility (not specific to Go) which auto-magically
  ; returns to the *next* instruction after its return address.
  3bad:       c3                      retq

  ; Main function body.

  ; Start by preparing the frame pointer.
  3bae:       55                      push   %rbp
  3baf:       48 89 e5                mov    %rsp,%rbp

  ; Store the arguments x, y, z into temporaries on the stack.
  3bb2:       48 89 7d e8             mov    %rdi,-0x18(%rbp)
  3bb6:       48 89 75 e0             mov    %rsi,-0x20(%rbp)
  3bba:       48 89 55 d8             mov    %rdx,-0x28(%rbp)

  ; Store zero (the default value) into a  temporary variable
  ; holding the return value at BP+8.
  3bbe:       48 c7 45 f8 00 00 00    movq   $0x0,-0x8(%rbp)
  3bc5:       00
  ; Re-load the arguments x and y from the stack.
  3bc6:       48 8b 55 e8             mov    -0x18(%rbp),%rdx
  3bca:       48 8b 45 e0             mov    -0x20(%rbp),%rax
  ; Compute x + y.
  3bce:       48 01 d0                add    %rdx,%rax
  ; Re-load z from the stack and compute x + y - z.
  3bd1:       48 2b 45 d8             sub    -0x28(%rbp),%rax
  ; Store the result value into the temporary variable
  ; for the return value.
  3bd5:       48 89 45 f8             mov    %rax,-0x8(%rbp)
  ; Re-load the return value from the temporary variable into
  ; a register.
  3bd9:       48 8b 45 f8             mov    -0x8(%rbp),%rax

  ; Restore the frame pointer, return.
  3bdd:       5d                      pop    %rbp
  3bde:       c3                      retq
```

This is very sad. GCC for other languages than Go is perfectly able to eliminate temporary variables. The following code would be just as correct:

```
FuncAdd:
   add %rdi, %rsi, %rax
   sub %rax, %rdx, %rax
   retq
```

(Disclaimer: these limitations can be lifted in a later version of `gccgo`.)

Summary of observations

The low-level calling convention used by the Go compiler on x86-64 targets is memory-heavy: arguments and return values are always passed on the stack. This can be contrasted with code generation by compilers for other languages (C/C++, Rust, etc) where registers are used when possible for arguments and return values.

The Go compiler uses dynamic registration (with a linked list of frame pointers) to prepare activation records for stack unwinding. This incurs a stack setup overhead on any function that calls other functions, even in the common case where stack unwinding does not occur. This can be contrasted with other languages that consider exceptions uncommon and implement table-driven unwinding, with no stack setup overhead on the common path.

Arguments and return values incur the standard memory costs of data types in Go. Scalar and struct types passed by value occupy their size on the stack. String and interface values use two words, slices use three. Because `error` is an interface type, it occupies two words.

Building an `error` value to return is usually more expensive than other values because in most cases this incurs a call to a vararg-accepting function (e.g. `fmt.Errof`).

The call sequence for vararg-accepting functions is the same as functions accepting slices as arguments, but the caller must also prepare the slice's contents on the stack to contain (a copy of) the argument values.

Go implements `defer`, a feature similar to `finally` in other languages. This is done by registering a callback in the current lightweight thread ("goroutine") at the beginning and executing the registered callbacks on every return path. This mechanism does not require heap allocation but incurs a small overhead on the control path.

Exceptions are thrown with `panic()` and caught with `defer` and `recover()`. Throwing the panic compiles down to a regular call to an internal function of the run-time system. That internal function is then responsible for stack unwinding. The compiler knows that `panic()` does not return and thus skips generating code for a return path. The mechanism to catch exceptions is fully hidden inside the pseudo-function `recover()` and does not require special handling for the code generator. Code generation makes no distinction between functions that may throw exceptions and those who are guaranteed to never throw.

The calling convention suggests there is a non-trivial trade-off between handling exceptional situations with `panic` vs. using `error` return values and checking them at every intermediate step of a call stack. This trade-off remains to be analyzed empirically in particular applications.

Next in the series:

- Measuring argument passing in Go and C++
- Measuring multiple return values in Go and C++
- Measuring errors vs. exceptions in Go and C++
- The Go low-level calling convention on x86-64 - New in 2020 and Go 1.15
- Errors vs exceptions in Go and C++ in 2020

Further reading

- JL Schilling — Optimizing away C++ exception handling — ACM SIGPLAN Notices, 1998

- David Chase — proposal: cmd/compile: define register-based calling convention, January 2017, last accessed July 2018
- Steven Huang — Golang Calling Convention (Chinese), August 2017, last accessed July 2018
- The Go Programming Language — Effective Go, last accessed July 2018
- Wikipedia — x64 Calling Conventions, last accessed July 2018
- Wikipedia — Exception handling implementation, last accessed July 2018
- Wikipedia — Structure of Call Stacks, last accessed July 2018

Copyright and licensing