Implementing faster defers

Dan Scales
Google

Defer Statement

- `defer` keyword followed by a function or method call
- The function pointer and arguments are evaluated immediately, but the call is deferred until the current function finishes
- The call is guaranteed to be called on any exit from the function, normal or panic
- New language construct in Go

Deferred Function Calls

```
func processFile(filename string, useStdin bool) error {
	var f *os.File; var err error

	if useStdin {
		f = os.Stdin
	} else {
		f, err = os.Open(filename)
		if err != nil { return err }
		defer f.Close()
	}
	src, err := ioutil.ReadAll(f)
	if err != nil { return err }
	…
}
```

Defer Performance Concerns

Date: Thu Sep 29 20:21:52 2016 -0400

runtime: remove defer from standard cgo call
The endcgo function call is currently deferred in case a cgo callback into Go panics and unwinds through cgocall. Typical cgo calls do not have callbacks into Go, and even fewer panic, so we pay the cost of this defer for no typical benefit.

This reduces the overhead for a no-op cgo call significantly:

```
name       old time/op  new time/op  delta
CgoNoop-8  93.5ns ± 0%  51.1ns ± 1%  -45.34%  (p=0.016 n=4+5)
```

Defer Performance Goal

Goal for defer optimization in Go 1.14:

- Make defer overhead low enough that programmers can use defer in any function whenever it is helpful!
- End the common belief that defers can be expensive

Outline

- Comparison of defer with related features in other languages
- Implementation of defer in Go 1.13
- Optimization of defer in Go 1.14
- Performance results & future optimizations

Related language constructs

- RAII (Resource Acquisition is Initialization) - C++
  - Destructor method is called when block that declared a variable ends, regardless of normal or abnormal exit
  - Example: `std::ofstream fs("example.txt");`
  - Statically analyzable, compiler knows exactly what destructors to call at each block end
  - Only applies to languages that are not garbage collected

Related language constructs (cont.)

- `try-finally` - Java, Javascript, Python, etc.
  - Statically analyzable, compiler knows exactly when to run ‘finally’ code
- Finalizer in Go
  - `runtime.SetFinalizer(obj, finalizer)`
  - Guarantees that a function will be called on an object before it is finally GC’ed
  - No guarantee that finalizer will run before exit or panic, just for avoiding slow leaks.

Advantages of Defer

- Improves maintainability: guarantees cleanup will happen even if more function exits are added
- Improves robustness: guarantees cleanup will happen even if there is a panic - especially important for recovered panics (e.g. long-running web server)
- Improves clarity - cleanup operation is described immediately following the relevant code
- Expresses conditional cleanup nicely

Implementing Go’s Defer

- Defer is completely dynamic (can be called inside a loop, conditional, etc.)
- A straightforward implementation must be handled by the Go runtime, in order to deal with the dynamic cases.
- The unusual cases (defer in a loop) can be expensive to handle
- But the common, simple cases are statically analyzable, and we would like these cases to be as efficient as C++ RAII

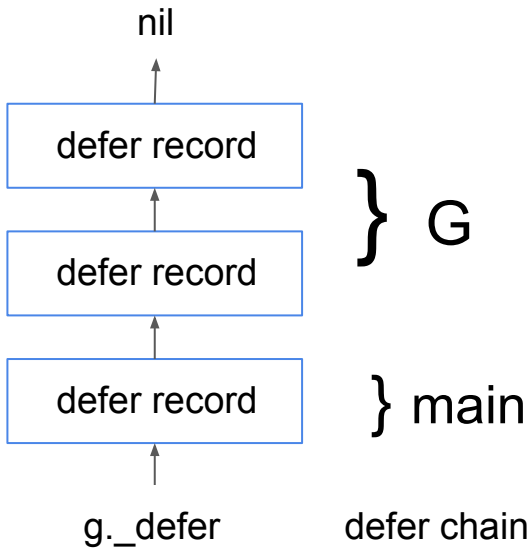
Implementation of defer in Go 1.13

```
func (p *Profile) Count() int {
        p.mu.Lock()
        defer p.mu.Unlock()
        if p.count != nil {
                return p.count()
        }
        return len(p.m)
}
```

```
func (p *Profile) Count() int {
        var r int
        p.mu.Lock()
        var d _defer
        d.fn = Unlock; d.arg1 = p.mu
        if runtime.deferproc(&d) == 1 {
                runtime.deferreturn()
                return r
        }
        if p.count != nil {
                …
        }
        r = len(p.m)
        runtime.deferreturn()
        return r
}
```

Defer chain

- Linked list of currently active defer records
  - Each defer record contains function pointer, defer args, pointer to stack frame
  - Each defer record may be on stack or heap (if defer is in a loop)
- **`runtime.deferproc`** initializes defer records and adds them to the chain
- **`runtime.deferreturn`** processes defer records in LIFO order from the defer chain, but only for the current frame



Defer chain & panics

- **`runtime.gopanic`** processes defer records in LIFO order as well:
  - Calls the appropriate function with the defer args
  - Continues until all defer records are processed, or one of the defer functions does a successful recover
  - If a successful `recover` occurs, runtime returns to the `deferproc` call whose defer did the recover with return value 1 - this causes a call to `deferreturn` and normal function return

Handling recovers

```
func (p *Profile) Count() int {
        var r int
        p.mu.Lock()
        var d _defer
        d.fn = Unlock; d.arg1 = p.mu
        if runtime.deferproc(&d) == 1 {
                runtime.deferreturn()
                return r
        }
        if p.count != nil {
                …
        }
        r = len(p.m)
        runtime.deferreturn()
        return r
```

Idea: make deferred calls directly in simple cases

- Only applicable for functions where no defer statement is in a loop
- At defer statements, compiler generates code to evaluate and store into stack slots the function pointer and any args, and to set `deferBits`
- At every exit from the function, compiler generates code to make each active defer function call (based on `deferBits`) in reverse order
- In normal execution, no defer record is created and no runtime function is called
- We call these “open-coded defers”

Implementation of defer in Go 1.14

```
func G {
        defer F1(a)
        if cond {
                defer F2(b)
        }
        body…
        return
}
```

```
func G {
        tmpF1 = F1; tmpA = a
        deferBits |= (1 << 1);
        if cond {
                tmpF2 = F2; tmpB = b
                deferBits |= (1 << 2);
        }
        body...
        if deferBits & 2 != 0 {
                deferBits &^= 2;
                tmpf2(tmpB)
        }
        if deferBits & 1 != 0 {
                deferBits &^= 1;
                tmpf1(tmpA)
        }
}
```

Implementation of defer in Go 1.14

```
func G {
        defer F1(a)
        if cond {
                defer F2(b)
        }
        body…
        return
}
```

```
func G {
        tmpF1 = F1; tmpA = a
        deferBits |= (1 << 1);
        if cond {
                tmpF2 = F2; tmpB = b
                deferBits |= (1 << 2);
        }
        body...
        if deferBits & 2 != 0 {
                deferBits &^= 2;
                tmpf2(tmpB)
        }
        if deferBits & 1 != 0 {
                deferBits &^= 1;
                tmpf1(tmpA)
        }
}
```

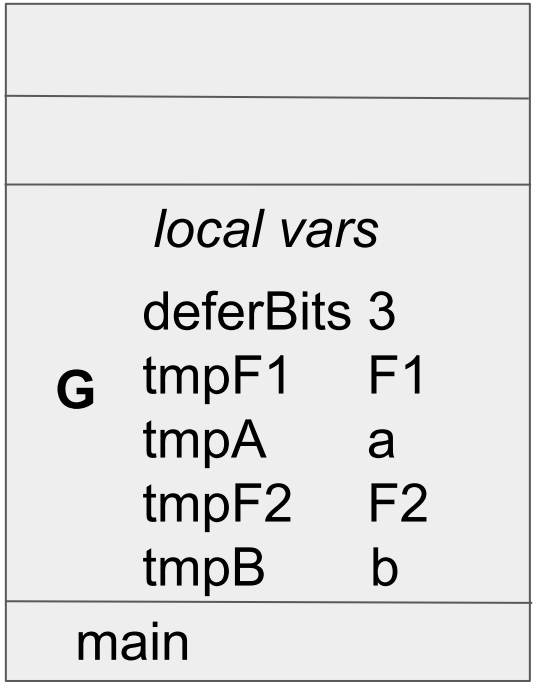
Implementation of defer in Go 1.14

```
func G {
        defer F1(a)
        if cond {
                defer F2(b)
        }
        body…
        return
}
```

```
func G {
        tmpF1 = F1; tmpA = a
        deferBits |= (1 << 1);
        if cond {
                tmpF2 = F2; tmpB = b
                deferBits |= (1 << 2);
        }
        body...
        if deferBits & 2 != 0 {
                deferBits &^= 2;
                tmpf2(tmpB)
        }
        if deferBits & 1 != 0 {
                deferBits &^= 1;
                tmpf1(tmpA)
        }
}
```

Implementation of defer in Go 1.14

```
func G {
        tmpF1 = F1; tmpA = a
        deferBits |= (1 << 1);
        if cond {
                tmpF2 = F2; tmpB = b
                deferBits |= (1 << 2);
        }
        body...
        if deferBits & 2 != 0 {
                deferBits &^= 2;
                tmpf2(tmpB)
        }
        if deferBits & 1 != 0 {
                deferBits &^= 1;
                tmpf1(tmpA)
        }
}
```



Compiler notes for open-coded defers

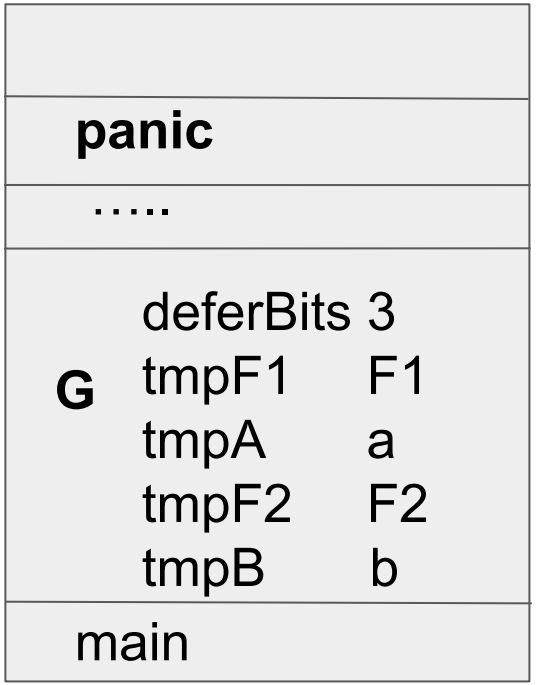
- We must ensure that the defer arguments are stored to their stack slots (not just kept in registers), so they are available in case of panic
- Similarly, updates to `deferBits` must be stored on the stack
- If all defers are unconditional, all checks on `deferBits` at exit are naturally optimized away
- For the simplest code, we limit the size of `deferBits` (currently 8 bits, could be more)

Panic processing for open-coded defers

- How to handle panics - we don’t have a defer record any more…
- Record in funcdata all the information about each defer, including stack slots where the function pointer and args are stored, etc.
  - “funcdata” is extra information in the Go binary that can be associated with any function
- To handle recovers, we need to add an extra stub of code for each function with open-coded defers
  - The stub calls `runtime.deferreturn()` and returns from the function

Panic processing for open-code defers (cont.)

Stack



G code

```
MOV …
…
...
RET
```

G.opendefer
funcdata entry

Offset of deferBits slot
Number of defers
For each defer:
  Offset of function pointer
  Number of args
  Offset/size of each arg

New algorithm for panic processing

- Scan the stack for frames that use open-coded defers, insert a record for each such frame in the defer chain
- Then iterate through the defer chain in usual LIFO order:
  - For each open-coded frame, read the funcdata for the function
  - Load the deferBits value, and iterate through all the active defers:
    - For each active defer, load the appropriate function pointer and args and execute the defer call
    - If a successful recover happened during a defer call, jump directly to the `deferreturn` stub in the function, which will process remaining defers in frame before returning to caller

Results

	Heap-allocated defers	Stack-allocated defers	Open-coded defers	Equivalent function call
1-arg function call	44.1ns	32.1ns	3.1ns	1.7ns
3-arg function call	48.9ns	34.3ns	5.5ns	4.4ns

- With open-coded defer in Go 1.14, the cost of a defer is now much closer to the cost of the function call itself, instead 8-18x more!

Results (cont.)

Noticeable improvements in some library code that uses defers:

- Invoking cgo callbacks from C back to Go
- Crypto

But other code may not improve, because defers have often already been removed in performance-sensitive functions!

Future optimizations

- Eliminate `deferBits` check for any unconditional defer (even when other defers are conditional)
- Use constant arguments directly in the defer calls at exit
- Do inlining of deferred calls
  - not currently done because inlining precedes defer statement processing in the compiler

Conclusion

- `defer` is a important language feature for improving maintainability and clarity
- defer has been optimized greatly in the last few years. The Go 1.14 work has made the common case much faster & closer to the efficiency of C++ RAII
- defer should be used whenever possible for code maintainability, and should **NOT** be avoided for performance reasons
- Thanks to Go team members at Google for their advice and help!