Errors vs. exceptions in Go and C++ in 2020

Why and how exceptions are still better for performance, even in Go

Raphael ‘kena’ Poss

December 2020

modified:	2020-12-01
subtitle:	Why and how exceptions are still better for performance, even in Go
slug:	go-errors-vs-exceptions-2020
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-errors-vs-exceptions-2020.html. Alternate formats: Source, PDF.

Introduction

Back in 2018, we compared empirically the performance overheads of Go and C++ with regards to the low-level calling convention: to pass arguments, to return multiple values, and when propagating exceptions.

Those results were obtained with Go 1.10 and Clang 6.0. Since then, compilers have evolved and we are now running with Go 1.15 and Clang 11.0. Have the findings changed since then?

The question of interest was:

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

In fact, as we will see below, the conclusions are stronger today than they were two years ago.

Methodology

The reader is first invited to familiarize themselves with the discussion of Go's low-level calling convention as compared with C++'s in the previous chapters of this series: as of 2018 and as of 2020. The 2018 analysis is still largely valid in 2020—the C++ calling convention did not change, and Go introduced a few optimizations on the implementation of `defer`.

There were three analyses previously:

- to measure argument passing and returning multiple values, we will use exactly the same methodology as the previous two analyses here and here.

  These analyses contain their benchmark source code, which is also provided at github.com/knz/callbench. The derivation of plots was entirely automated via Jupyter notebooks, which we can thus re-run.

- for the third analysis, which measures and compares the cost of error returns with that of throwing and catching exceptions, we are going to reuse the same methodology, but a diffence source code for the benchmark. The new source code has been uploaded to github.com/knz/callbench as well. This difference is motivated below.

We will also use an upgraded benchmark system, configured as follows:

- CPU: AMD Ryzen 7 3950X 3700.07MHz Family=0x17 Model=0x71 Stepping=0
- OS: FreeBSD 13.0-CURRENT r366805
- go version 1.15.5 freebsd/amd64
- C++ FreeBSD clang version 11.0.0 (llvmorg-11.0.0-0-g176249bd673)

Experimental results

All the generated data files and Jupyter notebooks can be downloaded here.

Findings for argument passing and returning multiple values

Here the findings are largely unchanged from 2018: for the selected benchmark code, the code generation by our selected compilers has not changed significantly.

Only the new hardware CPU influences the measurements: the new CPU has a higher clock frequency than previously, so for an equivalent mix of instructions we expect and observe a 10-40% raw performance improvement. Additionally, the Ryzen 7 3950X architecture has a somewhat more powerful superscalar execution unit than the 1800X, and is able to detect inter-instruction dependencies across more memory accesses. This causes more measurements artifacts when the workload is extremely small, and erases the performance differences between 1 and 5 function arguments or return values.

The main conclusions thus remain unchanged:

- Go uses memory to pass function arguments and return values. This makes Go code perform about twice slower than equivalent C++ code on function calls, where C++ compilers use register-based calling conventions instead.

  The particular CPU we used this year uses a micro-architecture able to more aggressively optimize memory-heavy code, and so the Go performance overhead relative to C++ is erased for very small functions; unfortunately, these optimizations become ineffective with more realistic, wider function bodies.

- In Go, memory is used for return values even when there is just one or two words worth of data. In these cases, a C++ compiler also typically uses registers. This choice in the Go compiler is unfortunate, as it mandates copying the return values through memory at every call level when functions return each other's value in a cascade. This cost thus multiplies the overhead of the memory-based return by the call depth, in a way that does not occur, or much less, with a typical C++ code generator.

- As previously, we can discuss what is the proportion of function calls relative to other types of code. Arguably, in computation-heavy algorithms, functions often get inlined so that calls disappear. In that case, it is possible for the Go compiler, assisted by the CPU micro-architecture, to reach similar instruction throughputs as the equivalent C++ code.

  However, we can also point out that Go still promotes the use of dynamic dispatch (via interface methods) as an idiomatic approach to software design. For example, string composition via `bytes.Buffer` interfaces with the `fmt` packages using interface calls (`io.Writer`), and the proportion of cross-interface calls is relatively high in that case compared to the computation load.

The reader is invited to review the more detailed summaries from the previous analyses, where the situation is presented in more detail.

Measuring errors vs panics: reminder and code tweak

This analysis calls for a benchmark which performs a "unit of work" in a loop, with the size of the loop configurable as "workload".

We are modeling the extremely common case where any error inside a unit of work aborts the entire workload: the error may or may not be generated inside the unit of work, but needs only be detected once for the entire outer workload loop.

In Go, we originally used the following code as benchmark:

```go
//go:noinline
func leafErr(arg int) (int, error) {
  if arg == 0 {
    // Unlikely error case.
    return 0, errObj
  }
  // Common non-error case.
  return id(arg), nil
}

//go:noinline
func workErr(work int) int {
  var n int
  for i := 0; i < work; i++ {
    val, err := leafErr(work)
    if err != nil {
      return 42
    }
    n += val
  }
  return n
}
```

In this code, `leafErr` represents the "unit of work", where errors can originate. The `workErr` represents the main work computation, where `leafErr` is called repeatedly.

(The `go:noinline` annotation is explained in more detail in the previous analysis.)

In the case where the `error` return is used, we need to check the `err` return on every iteration of the loop. If we used exception instead, we could factor the exception check out of the loop:

```go
//go:noinline
func workExc(work int) (res int) {
  // Exception check factored out of the loop:
  defer func() {
    if r := recover(); r != nil {
      res = 42
    }
  }()
  // Main workload follows:
  var n int
  for i := 0; i < work; i++ {
    n += leafExc(work + 1)
  }
  return n
}

//go:noinline
func leafExc(arg int) int {
  if arg == 0 {
    // Unlikely error case.
    panic(errObj)
  }
```

```
    // Common non-error case.
    return id(arg)
  }
```

(The accompanying C++ code is implemented similarly, using `std::tuple` for error returns and `try/catch/throw` for exceptions.)

Just looking at the shape of the source code, we can already suspect the findings that follow: there is less work inside the workload loop when using exceptions. So as the workload increases, the cost of setting up the "catch" logic gets amortized.

Indeed, this is exactly what was observed in 2018. (Summary and conclusions here.)

However, when running the same experiment in 2020 on our new test machine, with Go 1.15, the measurements are all over the place, the results appear chaotic, and the difference we observed last time is nearly entirely erased. What happened?

It turns out that the new CPU used to run the benchmark really has a much better micro-architecture. It is able to peek through the call from the `work` function to the `leaf` function, analyze inter-instruction dependencies across the call, and fully pipeline the processing of the `leaf` function across multiple iterations of the `work` loop. This effectively invalidates the model we used: in a realistic application, there is enough complexity in the code inside the "workload", when it contains function calls, that the CPU cannot pipeline loop iterations.

To restore a valid model, we should thus tweak the benchmark program by adding just enough additional complexity in the "unit of work". Here, we achieve that by adding just one extra level of intermediate call:

```
  //go:noinline
  func leafErr(arg int) (int, error) {
    if arg == 0 {
      // Unlikely error case.
      return 0, errObj
    }
    // Common non-error case.
    return id(arg), nil
  }

  //go:noinline
  func intermediateErr(arg int) (int, error) {
    return leafErr(arg)
  }

  //go:noinline
  func workErr(work int) int {
    var n int
    for i := 0; i < work; i++ {
      val, err := intermediateErr(work)
      if err != nil {
        return 42
      }
      n += val
    }
    return n
  }
```

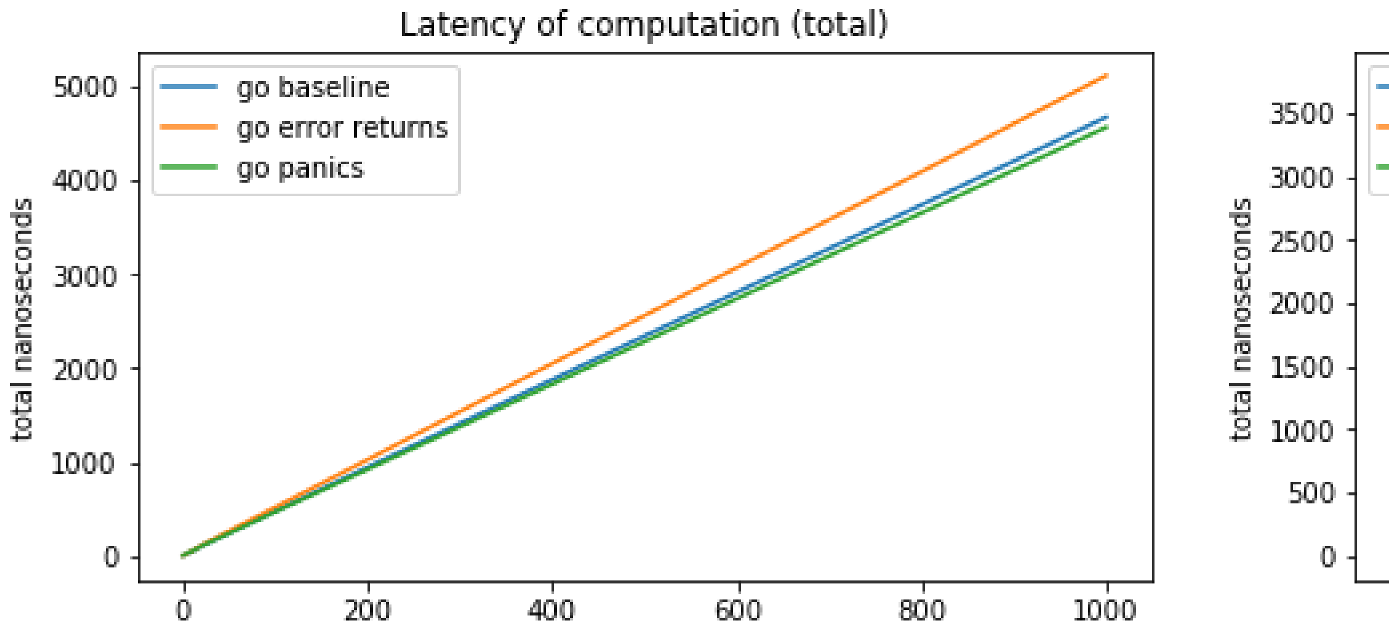
(Likewise for the benchmark using exceptions.)

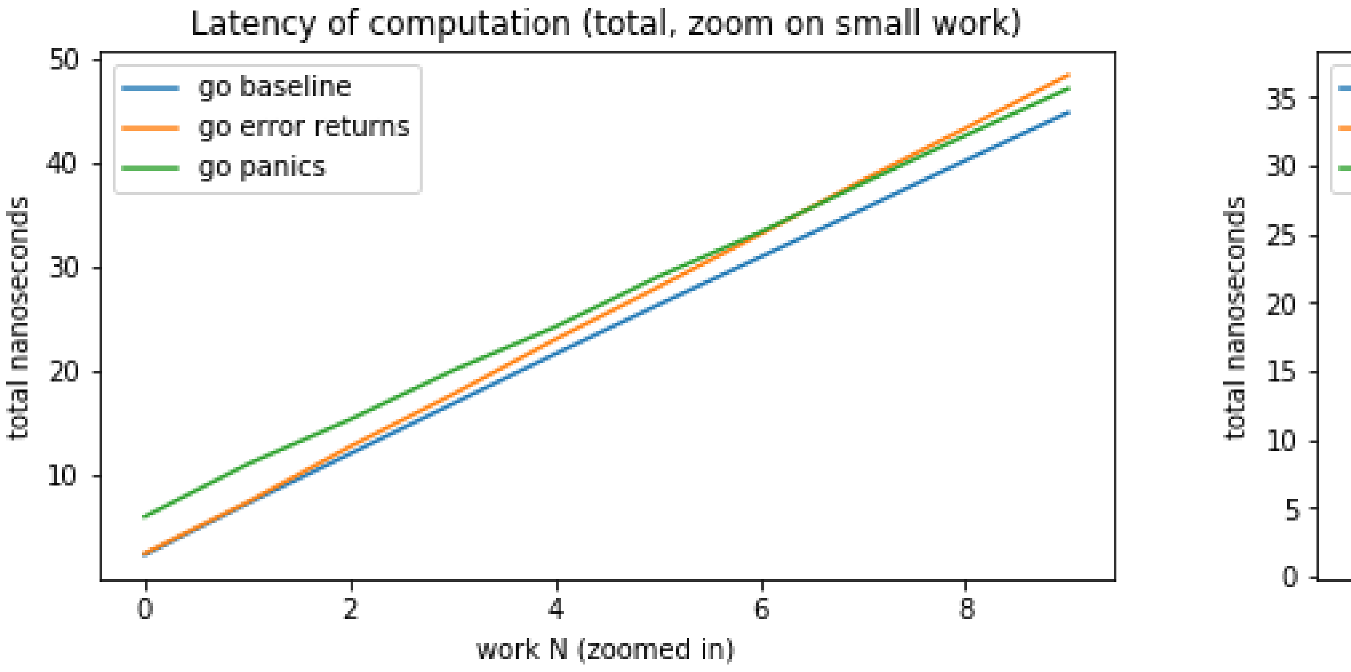
This program, at least on the CPU considered, is "complex" enough to stop the superscalar unit from peeking across the function call. The resulting code then behaves regularly in measurements.

Findings for error returns vs exceptions

The Jupiter notebook for this experiment contains the data processing steps and the logic to generate comparison plots, with some detailed analysis of the intermediate results.

The key result is the following:





What we see here:

- For an equivalent workload (e.g. 1000 "units of work"), the Go code is 40% slower than the equivalent C++ code. This is generally expected as Go is much more memory-heavy than

C++.

- Regardless of the language considered, the cost of setting up an exception "catch" can be reliably amortized across an entire workload: the performance of code using exception-based error reporting asymptotically converges to become the same as code using no error handling whatsoever.

  In comparison, code that propagates errors as return values is systematically slower as the workload increases; between 1-3% for C++, up to 4-10% for Go.

  This is one of the two main results.

- The cost of setting up an exception "catch" is not entirely negligible. Back in 2018, the mandatory fixed cost of the `defer` setup and testing `recover()` in the `work` function was 50ns; such that it took about 500 nanoseconds worth of "workload" to fully offset this one-time cost.

  In 2020, thanks to the `defer` optimizations in Go 1.15, the fixed overhead was greatly decreased; down to just a few instructions and, on the current test hardware, just 11ns. It now takes merely 35 nanoseconds worth of "workload" to fully offset the one-time cost.

  This is the second main result.

Summary and conclusions

The main conclusions from the previous iteration of this experiment remain, and are even strengthened.

The particular way the Go compiler generates code makes error signalling via return values incur a non-negligible performance overhead — between 4% and 10% in our tests — compared to code which only returns its main results.

This price is mostly incurred by Go's usage of memory to pass return values (a choice unlikely to be revisited any time soon, as per the discussion on a proposal to change it).

When using exceptions instead to signal errors ("panics" in Go), this overhead is eliminated from the computation code in the common case when errors do not occur. However! The fixed cost to set up an exception handler is non-zero in Go, *even when exceptions are uncommon/rare.* Therefore, exception handling in Go is only advantageous performance-wise relative to error returns when there is enough work "under" the error handler to offset this fixed cost.

(In comparison, C++ code generators commonly apply the "zero cost abstraction" principle and ensure that exceptions do not cost anything unless they actually occur — so adding a `try/catch` block does not reduce performance in any way.)

Previously, in Go 1.10, this fixed cost was non-negligible, climbing upwards of dozens of nanoseconds. Thanks to recent improvements in the Go compiler however, as well as general improvements in CPU micro-architectures, this cost has been greatly reduced in 2020. So much, in fact, that there is less need for carefully considering how much work is performed under a common error handler to decide between one approach or the other.

The empirical record for Go "best software practices" for error signalling *when errors are rare/uncommon* thus evolves as follows:

Use case	**Error returns**	**Exceptions ("panics" in Go)**
Tutorial-level examples with little relevance to the real world	Pros • seemingly simple to use • simple to explain Cons • none	Pros • none Cons • harder to explain • confusing for beginners
Real-world computations, with mixes of computations and abstraction, where performance may or may not matter	Pros • none Cons • **makes the code slower**, increases latency and reduces throughput • boilerplate, repetitive code • cumbersome to write • lots of noise when reading and maintaining the code	Pros • **makes the code faster** • condenses the code, makes it easier to read and maintain over time Cons • needs discipline and use `defer` consistently to ensure RAII

Further reading

Also in the series:

- The Go low-level calling convention on x86-64
- 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

Copyright and licensing