Measuring multiple return values in Go and C++

Raphael kena Poss

August 2018

Contents

> **Note**
> The latest version of this document can be found online at https://science.raphael.poss.name/measuring-multiple-return-values-in-go-and-cpp.html. Alternate formats: Source, PDF.

Introduction

The following document investigates the performance of returning multiple values from functions in Go, and compare that to C++.

This is a follow-up to the Go calling convention on x86-64 and Measuring argument passing in Go and C++.

We focus here on just one of the aspects of the Go calling sequence: returning multiple values functions and how the number of return values influences performance.

Further analysis of the other aspects of calling sequences (return values, errors, exceptions) are still out of scope and will be the topic of a later analysis.

> **Note**
> This document was designed as Jupyter Notebook. The notebook and accompanying data files can be downloaded here.

Multiple return values in the calling convention

In the previous article on the Go calling convention, I explained that Go returns values on the stack. The first return value is located at address SP+32 (0x20 + SP).

In the standard calling convention for x86-64, the one used by most C++ compilers, the first two return values are passed into registers %rax and %rdx. For example, the function

```
std::tuple<long,long> fret(long i) { return {i, i}; }
```

compiles to:

```
fret(long):
        movq    %rdi, %rax
        movq    %rdi, %rdx
        retq
```

(The first argument in %rdi, the two return values in %rax and %rdx.)

When a function returns 3 words worth of values or more, a different calling convention is used:

- the caller allocates space for the return values on the stack.
- the caller passes the address for the return value as first implicit argument to the function, in register %rdi.
- the callee uses the address in the first argument (%rdi) as base address to write return values.

- the address of the return value is also returned in %rax.

This calling convention behaves as if a function originally written like this:

```
std::tuple<long,long,long> fret(long i) { return {i, i, i}; }
```

was actually written like this:

```
std::tuple<long,long,long>* ret
fret(std::tuple<long,long,long>& ret, long i) {
  ret = {i, i, i};
  return &ret;
}
```

The motivation for having the caller allocate the space *and pass its address as argument* (instead of relying on an offset relative to the stack pointer) is to **avoid copying return values when there are multiple layers of calls**: each intermediate function can pass the address of its caller's return space as first argument to its callee. In the function where the return value is actually constructed, it can directly construct it in the final destination space.

Obviously, this optimization cannot be utilized by Go.

From a performance perspective, we can expect the machine code generated for C++ to be faster at returning 1 or 2 values than the equivalent Go code, because they are passed using registers. For 3 or more values, they should be about the same.

Let's check this experimentally.

Experimental setup

The source code for the experiments can be found here: http://github.com/knz/callbench. There are multiple experiment sources in that repository, today we are focusing on the `ret` experiments: comparing the result of returning varying number of return values from Go and C++ functions.

Measuring the cost of just one return

The functions we are going to measure look like this:

- in Go,

```
//go:noinline
func f4r(i int) (int,int,int,int) { return i,i,i,i }

func BenchmarkRet4(b *testing.B) {
   val := 1
   for i := 0; i < b.N; i++ {
     r, _, _, _ := f4r(i)
     val += r
   }
   CONSUME(b, val);
}
```

- in C++,

```
__attribute__((noinline))
std::tuple<long,long,long,long> f4r(long i) { return {i,i,i,i}; }

void BenchmarkRet4(B* b) {
  long val = 1, N = b->N;
  for (long i = 0; i < N; i++) {
    NOP(i);
    auto t = f4r(i);
    val += std::get<0>(t);
  }
  CONSUME(b, val);
}
```

There are 20 variants of these benchmarks, from 1 to 20 return values. They are generated by the accompanying `gen_ret.sh` scripts.

In every case, the function replicates its arguments for each return value. The caller discards all but the first.

The C++ code uses `std::tuple` because C/C++ do not have a native syntax to return multiple return values. As the example code above demonstrates, `std::tuple` is the next best thing. In C it is common to return multiple return values via a plain struct; under the hood this is exactly what `std::tuple` does under the hood: the values are packed next to each other in an auto-generated struct.

The functions are marked as `noinline` to ensure they are not inlined in the benchmark kernel, and the call sequence in machine code eliminated, by optimizations.

The benchmark kernel uses in Go the standard `go test -bench` infrastructure. In C++, it uses the library cppbench which re-implements the `go test -bench` infrastructure in C++, so that we are measuring the same things in the same way in both languages.

See the previous article for details about `CONSUME()` and `NOP()`.

Measuring the cost of a stack of returns

The section above claims that the benefit of passing the address of the return struct in code generated from C++ is to avoid copying the data. What is the performance benefit exactly?

What we expect to see is the following:

- if there is a call stack of N functions that all have the same M return value types,
- then the call latency for the first N-1 intermediate function calls is the same regardless of the value of M (because the intermediate function calls do not need to do work in proportion to M).

In comparison, Go *does* need to copy the M values at every step, so for code generated from Go we'll see the N-1 intermediate function calls have a performance cost increasing with the value of M.

We'll evaluate this as follows:

- in Go,

```
//go:noinline
func f4rmulti(d,i int) (int,int,int,int) {
  if d == 1 { return i,i,i,i }
```

```
  return f4rmulti(d-1, i)
}

func benchMultiRet4(d int, b *testing.B) {
   val := 1
   for i := 0; i < b.N; i++ {
     r, _, _, _ := f4rmulti(d, i)
     val += r
   }
   CONSUME(b, val);
}
func BenchmarkMultiRet1_4(b *testing.B) { benchMultiRet4(1, b); }
func BenchmarkMultiRet20_4(b *testing.B) { benchMultiRet4(20, b); }
```

- in C++,

```
__attribute__((noinline))
std::tuple<long,long,long,long> f4rmulti(long d, long i) {
  NOP(d);
  if (d == 1) return {i,i,i,i};
  return f4rmulti(d-1, i);
}

void benchMultiRet4(long d, B* b) {
  long val = 1, N = b->N;
  for (long i = 0; i < N; i++) {
    NOP(i);
    auto t = f4rmulti(d, i);
    val += std::get<0>(t);
  }
  CONSUME(b, val);
}
void BenchmarkMultiRet1_4(B* b) { benchMultiRet4(1, b); }
void BenchmarkMultiRet20_4(B* b) { benchMultiRet4(20, b); }
```

What this does is define a recursive function `fNrmulti` that performs a stack of call with the depth specifies by its first argument. For example, the call `f4rmulti(10, 123)` will make `f4rmulti` call itself 10 times. The 10th activation is the one that actually constructs the return value.

As before we use the pseudo-function `NOP` to ensure the compiler is unable to peek into the depth variable and flatten the recursion.

We will also be careful to compile with `-fno-optimize-sibling-calls` in order to prevent tail call optimization from kicking in and transforming the call stack into a simple loop.

We'll run with depth 1 and depth 20. What we will want to see is what happens if we subtract the latency of depth 1 from that of depth 20. How expensive are the 19 remaining calls? With C++ we'd expect this to be independent of the number of return values, whereas with Go it should be proportional.

Running the benchmarks

We run all these 120 benchmarks as follows:

- `cd go; make ret_go.log`
- `cd cpp; make ret_cpp.log`

Then we copy the two log files into the directory of the Jupyter notebook.

The specific measurements illustrated in the rest of this document were obtained in the following environment:

- CPU: AMD Ryzen 7 1800X 3593.33MHz Family=0x17 Model=0x1 Stepping=1
- OS: FreeBSD 12.0-CURRENT r336756
- go version 1.10.3 freebsd/amd64
- C++ FreeBSD clang version 6.0.1 (tags/RELEASE_601/final 335540)

Data preparation

We will use the same data preprocessing as the previous article. Check the corresponding section there for an explanation.

```
def load(fname):
    data = [x for x in open(fname).read().split('\n') if x.startswith('Benchmark')]
    data = [x[9:] for x in data]
    return data

data = ['Go/' + x for x in load('ret_go.log')] + \
       ['Cpp/' + x for x in load('ret_cpp.log')]
print ("number of results:", len(data))
print ("example result row: %r" % data[0])

number of results: 120
example result row: 'Go/Ret1          t1000000000t         2.54 ns/op'

import re
r = re.compile('^(\S+)\s+\S+\s+(\S+)\s+.*')
data = [m.groups() for m in [r.match(x) for x in data] if m is not None]
print ("example result row: %r" % (data[0],))

example result row: ('Go/Ret1', '2.54')

data = [(x[0], float(x[1])) for x in data]
print("example result row: %r" % (data[0],))

example result row: ('Go/Ret1', 2.54)

def filterdata(pattern):
    r = re.compile(pattern)
    matchvals = [(r.match(x[0]), x[1]) for x in data]
    xvals = [int(i[0].group(1)) for i in matchvals if i[0] is not None]
    yvals = [i[1] for i in matchvals if i[0] is not None]
    return xvals, yvals

xgo, ygo = filterdata('Go/Ret(\d+)')

# Really the X values are the same for all series.
xvals = xgo
_, ygomulti1 = filterdata('Go/MultiRet1_(\d+)')
_, ygomulti20 = filterdata('Go/MultiRet20_(\d+)')
_, ycpp = filterdata('Cpp/Ret(\d+)')
_, ycppmulti1 = filterdata('Cpp/MultiRet1_(\d+)')
_, ycppmulti20 = filterdata('Cpp/MultiRet20_(\d+)')
```

Data analysis

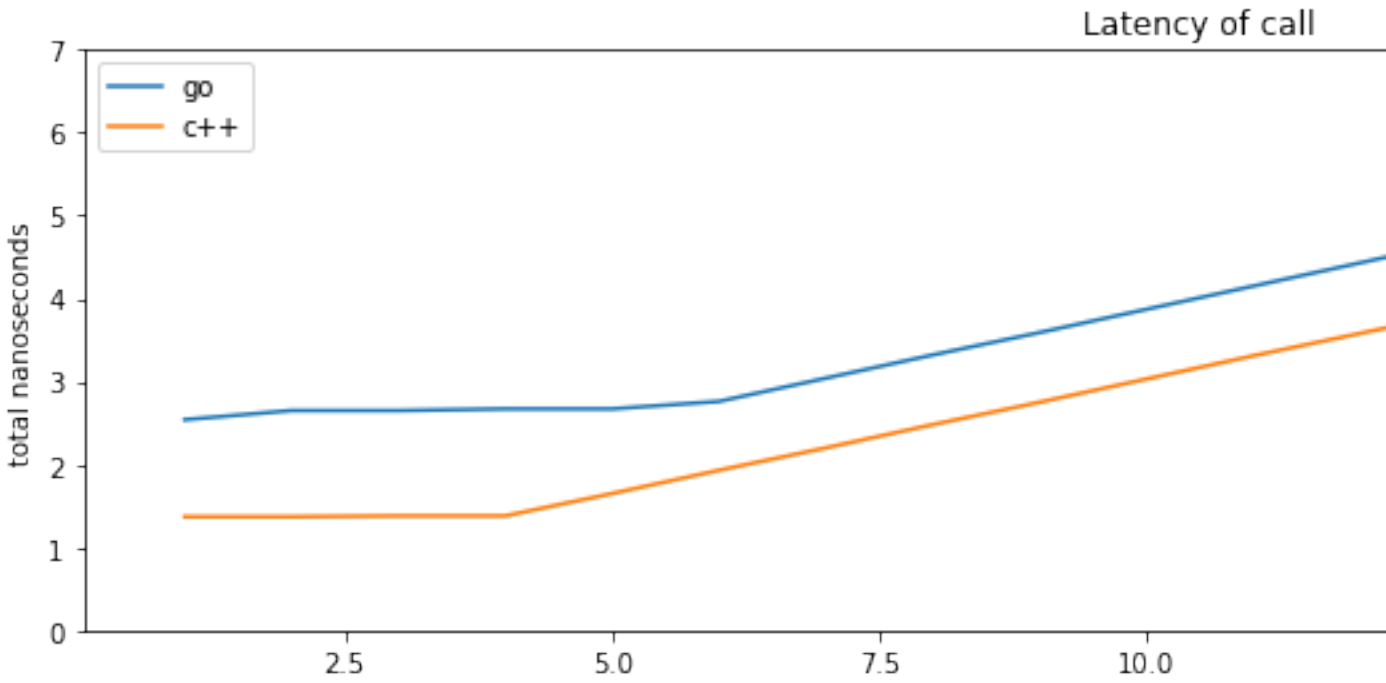
We'll use `matplotlib` for plotting. This needs to be initialized first:

```
%matplotlib inline

import matplotlib
import matplotlib.pyplot as plt
```

Analyzing the cost of the return itself

We can compare our benchmark results by plotting their x- and y-series extracted above:

```
plt.figure(figsize=(15, 4))
plt.plot(xvals, ygo, label='go')
plt.plot(xvals, ycpp, label='c++')
plt.title('Latency of call')
plt.ylabel('total nanoseconds')
plt.ylim(0, 7)
plt.legend()

plt.show()
```



What does this tell us?

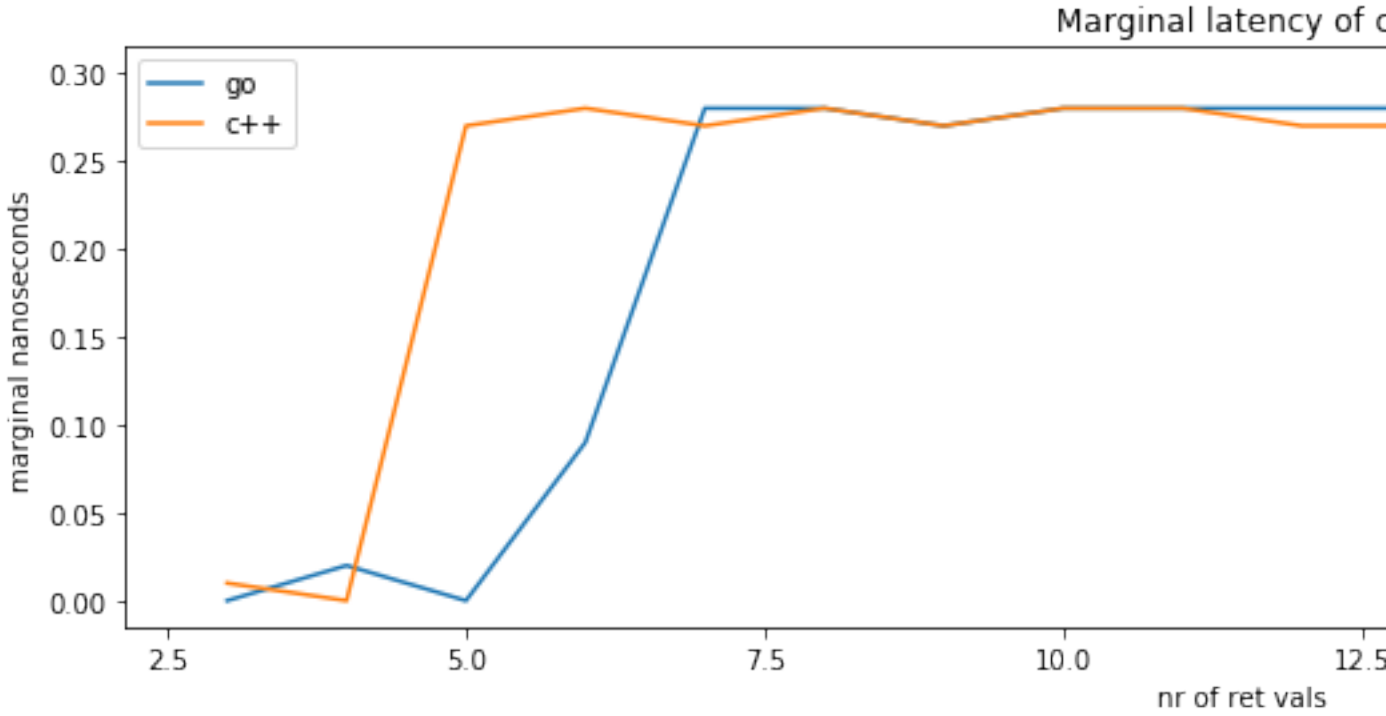
As expected, the code generated from C++ is faster than Go. **Between 1 and 5 return values (the most common cases in real-world code), the Go calling convention is about 60% slower than that of C++.**

However the difference remains constant as the number of return values increases, because beyond 2 return values both languages use memory.

What is a bit more interesting, this time, is that the call latency does not really change *in either language* until 3 (C++) or 5 (Go) or more values are returned. This is true despite the observation that the generated code (in both languages), in these cases, contain memory operations in proportion to the number of return values.

Let's dive deeper. What is the marginal cost of adding more return values?

```
xvalsm = xvals[2:]

def marginals(yvals):
    return [y - yvals[i] for (i, y) in enumerate(yvals[1:])]

ygom = marginals(ygo[1:])
ycppm = marginals(ycpp[1:])

plt.figure(figsize=(15, 4))
plt.plot(xvalsm, ygom, label='go')
plt.plot(xvalsm, ycppm, label='c++')
plt.title('Marginal latency of call')
plt.xlabel('nr of ret vals'); plt.ylabel('marginal nanoseconds')
plt.legend()
plt.show()
```



In C++, the marginal cost to go from 1 to 2 return values appears to be zero, although the generated code contains one more instruction in the called function.

Like in the previous analysis, we are observing the effect of multiscalar issue: during the benchmarking loop where the function is called many times, the instructions from one call are still processing while the processor is starting the next call. This is a valid hardware optimization because the return value is not used as a carried dependency from one iteration of the loop to the next.

Even though Go uses memory, in this particular CPU the memory operations are candidate for superscalar issue too, so there is the same benefit from 1 to 2 arguments.

When the C++ code goes from 2 to 3 return values, it starts using memory. This is relatively more expensive, hence a jump of 0.25ns.

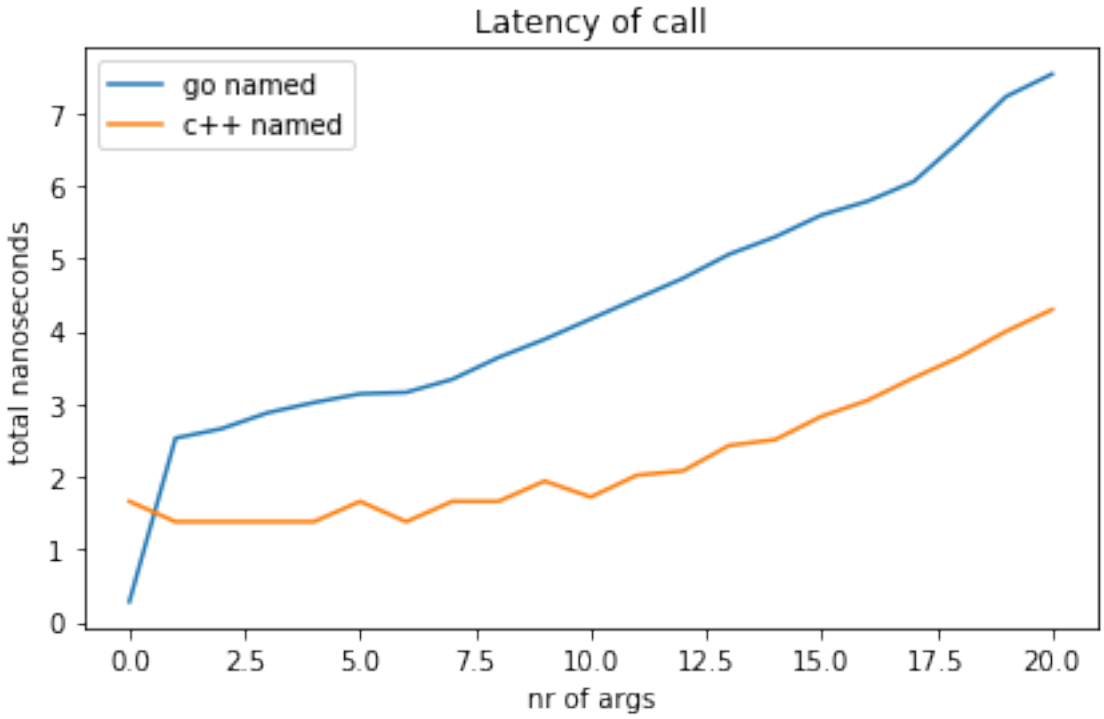
The Go code evolving from 2 to 3 return values has already paid the price of going to memory for 2 return values, so the marginal increase from 2 to 3 is negligible. Again we are seeing the benefit of superscalar issue.

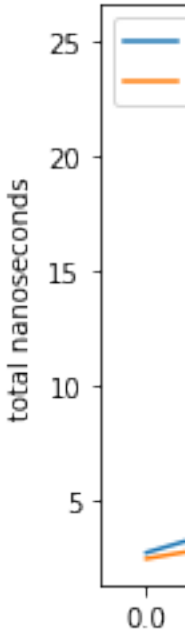
The effects of superscalar issue here are quite pronounced. They were not as pronounced in the previous analysis. Here's why. In the previous analysis, there was a computation inside the function (adding the arguments together) so there was a dependency from all the register argument to the result, through all the instructions that computed the result. In this analysis, *although the function computes many return values, only the first result value is used in the benchmarking loop*. So the CPU only has to wait for the first return value to be computed to issue the `add` in the benchmarking loop and start the next iteration. The remaining return values can be computed in parallel with the first return values of the next call.

This optimization remains available until the pipeline is full, which according to this experiment happens after 5 return values.

Aside: what of the limitation in the previous analysis?

In the previous analysis, we observed this:





And I then explained that superscalar instruction issue is usually optimized for register-register computations, and prevented by dependent memory accesses (i.e. when subsequent instructions use the same memory location). Even when the micro-architecture can overlap memory accesses,

the max number of in-flight memory instructions is typically smaller than that of register-register instructions.

This would tend to suggest that we should see the marginal cost always be greater than zero when increasing the number of return values beyond 3, since memory is used in both cases.

Insted we observe that the marginal cost is about zero until 5 return values in Go. Was I wrong before?

Actually, what we are observing here is cleverness in the issue unit of this particular CPU.

In the previous experiment, there was a read-after-write carried dependency between the memory operations across iterations of the benchmarking loop: the loop counter was duplicated (written) to memory *into* each call, then the argument values were all read *inside* the call to compute the return value. The next iteration erased the values (to replace them by the new iteration counter) but only after they had been read by the previous iteration.

A superscalar issue usually really does not like read-after-write (RAW) dependencies on memory. This is because if the CPU decided to start iteration N+1 while iteration N is still running, it could erase the values in memory for iteration N+1 before iteration N had a chance to read them. Instead of running this risk, RAW memory dependencies are avoided by preventing the superscalar issue.

(The reason why it works when using registersi.e. RAW dependencies do not prevent superscalar issuesis that RAW dependencies with register-register instructions are resolved using register renaming. Renaming is only available for registers, though.)

Now, in this new analysis we are using a benchmark which *does not use the returned values* (except the first). So for every return value but the first, we really have a write-after-write (WAW) dependency on the memory locations. This particular CPU sees the WAW dependency, sees that there is no read of the value before the next write, and decides that it is safe to issue the instructions concurrentlya reorder buffer at the end of the pipeline is available to ensure that the final value in memory will be the right one, independently of the actual execution order.

So the CPU has to wait for the instruction(s) that compute the 1st return value to complete before starting the next benchmark iteration/call (because there is still a RAW dependency on that location), but when that has completed, it can start the next call while the remaining return values are computed in the first iteration.

Hence the observation: the benchmarking loop in this anlaysis is better amenable to superscalar execution than that of the previous analysis, because we are not consuming all the return values.

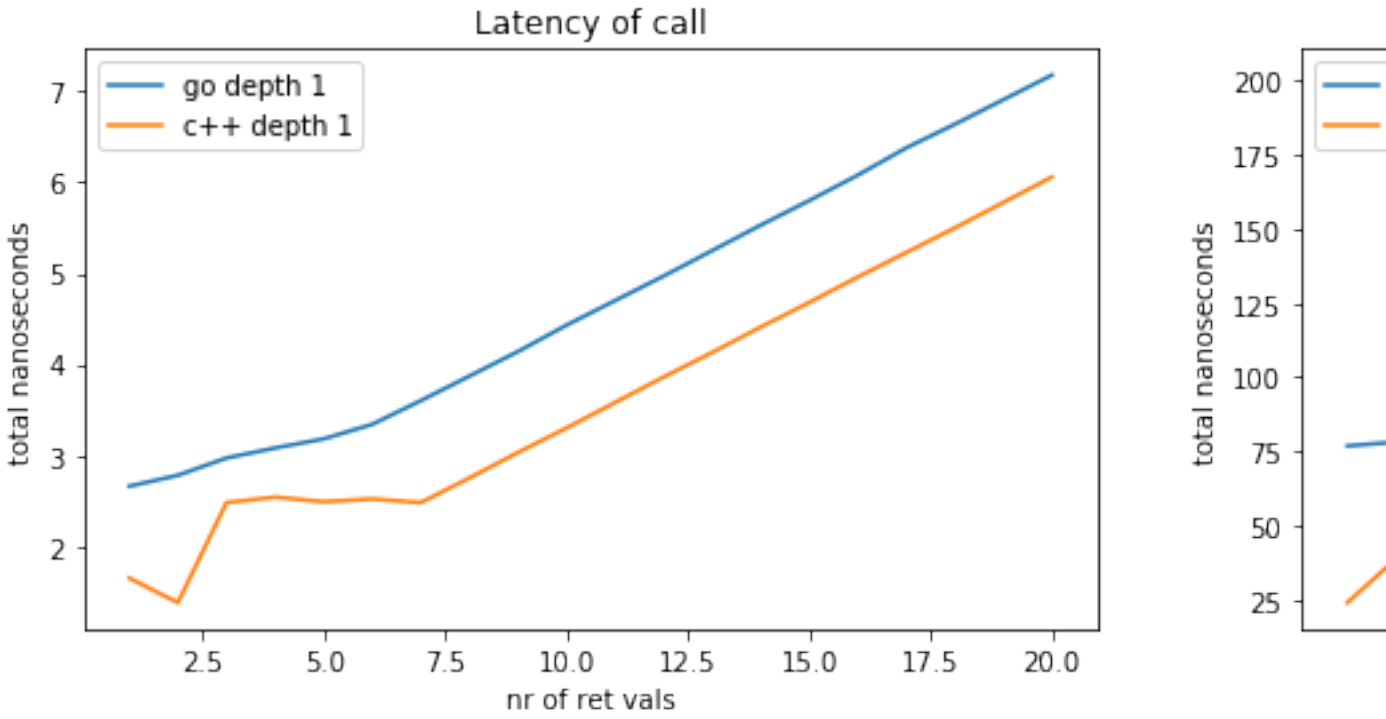
Analyzing intermediate calls

When there is a stack of function calls, where a multi-valued return occurs through multiple intermediate calls, the standard calling convention for x86-64 avoids copying, whereas Go has to copy in all cases.

We can look at this with the following plots:

```
plt.figure(figsize=(15, 4))
plt.subplot(1,2,1)
plt.plot(xvals, ygomulti1, label='go depth 1')
plt.plot(xvals, ycppmulti1, label='c++ depth 1')
plt.title('Latency of call')
plt.xlabel('nr of ret vals'); plt.ylabel('total nanoseconds')
plt.legend()
```

```
plt.subplot(1,2,2)
plt.plot(xvals, ygomulti20, label='go depth 20')
plt.plot(xvals, ycppmulti20, label='c++ depth 20')
plt.title('Latency of call')
plt.xlabel('nr of ret vals'); plt.ylabel('total nanoseconds')
plt.legend()

plt.show()
```



We're replotting here the cost of 1 function call, because we are now looking at a different function (`fNrmulti`) than the previous section (that used `fNr`) and thus the baseline is different.

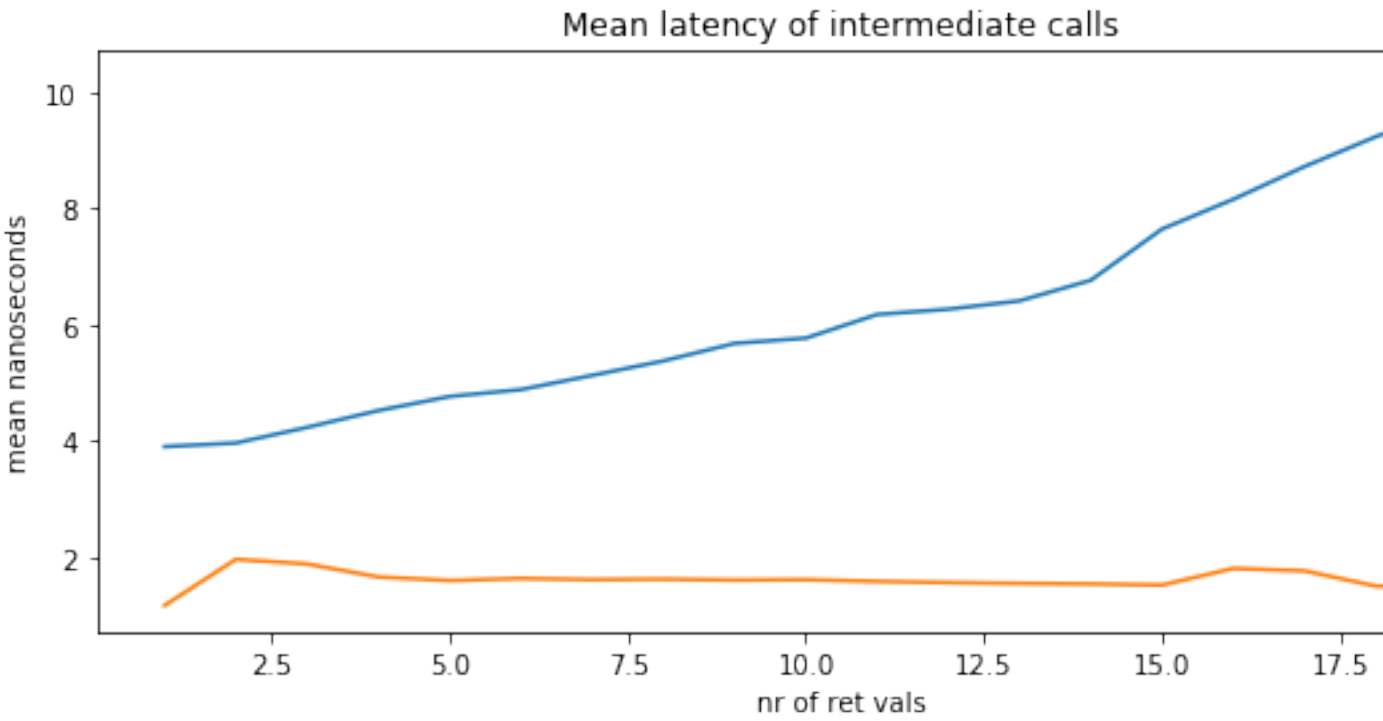
The latency of 1 call is somewhat higher here than in the previous section, because both the C++ and Go code must now prepare an activation record on the stack for the recursively called function. Also there is 1 more argument to determine the call depth.

Then on the right, we are plotting the cost of a stack of 20 calls.

The cost to perform 20 recursive calls in C++ seems to be much lower, as expected.

What is the individual cost of the intermediate calls? To measure this, we take the latency of the 20 recursive calls, and we subtract the latency of the final call which constructs the return values. This gives us the latency of the 19 remaining calls, which we divide by 19 to compute the mean latency per intermediate call.

```
ygomulti19 = [(y20-y1)/19. for (y1, y20) in zip(ygomulti1, ygomulti20)]
ycppmulti19 = [(y20-y1)/19. for (y1, y20) in zip(ycppmulti1, ycppmulti20)]

plt.figure(figsize=(10, 4))
plt.plot(xvals, ygomulti19, label='go')
plt.plot(xvals, ycppmulti19, label='c++')
plt.title('Mean latency of intermediate calls')
plt.xlabel('nr of ret vals'); plt.ylabel('mean nanoseconds')

plt.show()
```

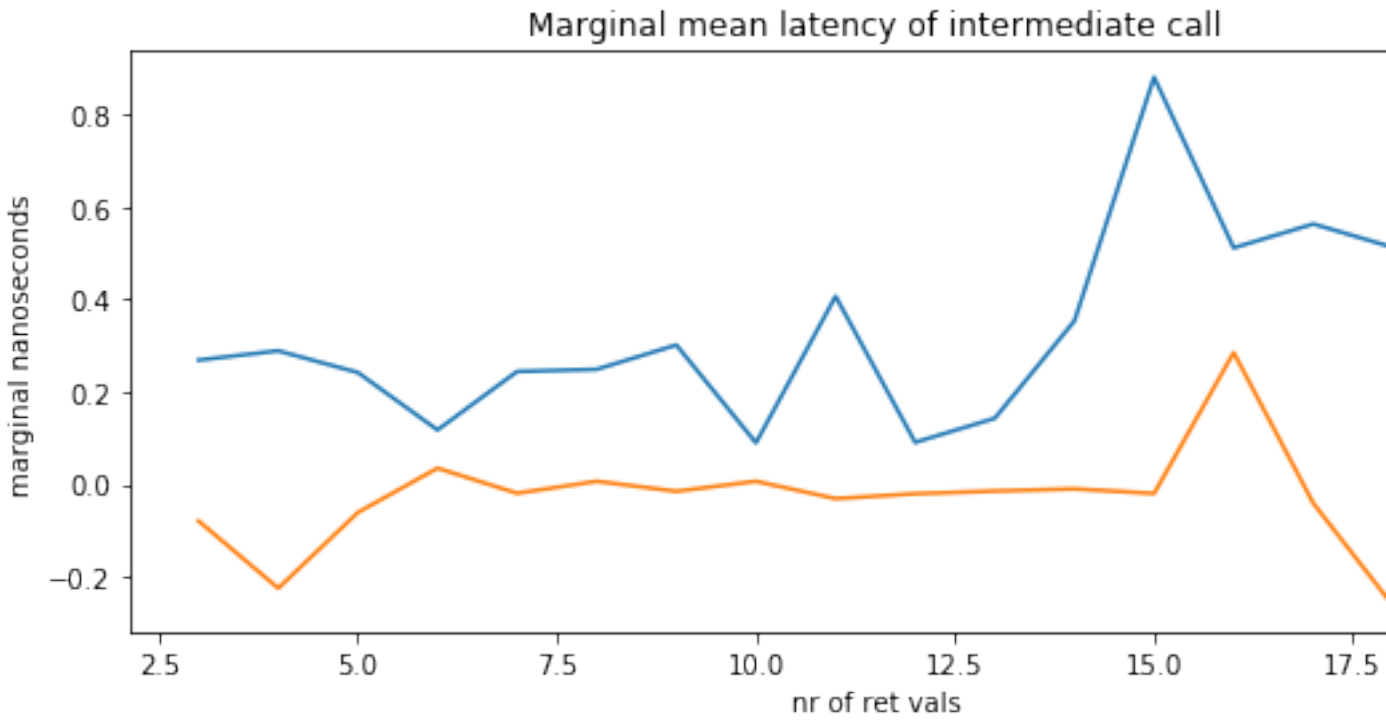


This confirms what the analysis of the calling convention told us already: the intermediate calls in C++ do not incur extra costs when the number of return values (or the size of a struct, if a struct is returned by value) increases.

It does in Go, though.

By how much exactly? Let's look at the marginal cost of adding one more return value each time:

```
xvalsm = xvals[2:]
def marginals(yvals):
    return [y - yvals[i] for (i, y) in enumerate(yvals[1:])]

ygom = marginals(ygomulti19[1:])
ycppm = marginals(ycppmulti19[1:])

plt.figure(figsize=(10, 4))
plt.plot(xvalsm, ygom, label='go')
plt.plot(xvalsm, ycppm, label='c++')
plt.title('Marginal mean latency of intermediate call')
plt.xlabel('nr of ret vals'); plt.ylabel('marginal nanoseconds')
plt.legend()
plt.show()
```



The marginal cost in intermediate calls is zero in C++ until about 15 return values, whereas it is strictly greater than zero in Go.

In other words, **the size/number of return values has a multiplicative effect on the latency of deep call stacks in Go**, whereas it has no effect in C++.

At 16 return values, we observe a spike in the marginal cost (paid back at 18 return values whne there's a corresponding negative cost, i.e. a win). I do not have a satisfying explanation for this, but I suspect cache effects. I consider this to be inconsequential since real-world code tends to use fewer return values anyway.

Summary and conclusions

On an off-the-shelf, relatively modern x86-64 implementation by AMD, the calling convention of Go makes Go function calls perform 60% slower than the equivalent C++ code to return fewer than 5 values (or 5 words worth of values, e.g. when enclosed in structs).

This performance difference is caused by the choice of Go to use memory to pass return values (a choice unlikely to be revisited any time soon, as per the discussion on a proposal to change it).

Moreover, if multiple functions call each other and return each other's return value(s), the size of the return values multiplies the latency of the entire call stack in Go, because of the data copying in each intermediate call, whereas it has no impact on code generated from C++.

This performance difference is caused by the choice of Go to *use an address relative to the stack pointer for the return values* (this is a different choice than the first). In contrast, the standard calling convention recommends passing the address to the return values as implicit argument, so that this return address can be forwarded to further callees without data copying. This optimization is available to C++ today, and might be available in a future version of Go.

As in the previous analysis, one should take these observations with a grain of salt: the cost of function calls is usually amortized by sufficient code "inside" the function, or by inlining of calls. However, this grain of salt in favor of Go unfortunately dissolves when the function body is not quite large enough to offset the cost of the calling sequence, and the function cannot be inlined. This commonly happens with recursive functions or dynamic dispatch ("virtual calls" in C++ / "interface calls" in Go). There the standard calling convention, and thus C++, gains a clear advantage. Again, I plan to revisit these topics in a later analysis.

Also in the series:

- The Go low-level calling convention on x86-64
- Measuring argument passing in Go and C++
- Measuring errors vs. exceptions in Go and C++

Copyright and licensing

SC fingerprint: `fp:2PhG3tgymvupfypVIpDsTwbeixaGZQsNmopq_axbPzrJqw`