Golang bcc/BPF Function Tracing

31 Jan 2017

In this post I'll quickly investigate a new way to trace a Go program: dynamic tracing with Linux 4.x enhanced BPF (aka eBPF). If you search for Go and BPF, you'll find Go interfaces for using BPF (eg, gobpf). That's not what I'm exploring here: I'm using BPF to instrument a Go program for performance analysis and debugging. If you're new to BPF, I just summarized it at linux.conf.au a couple of weeks ago (youtube, slideshare).

There's a number of ways so far to debug and trace Go already, including (and not limited to):

- Debugging with gdb and Go runtime support.
- The go execution tracer for high level execution and blocking events.
- GODEBUG with gctrace and schedtrace.

BPF tracing can do a lot more, but has its own pros and cons. I'll demonstrate, starting with a simple Go program, hello.go:

```
package main

import "fmt"

func main() {
        fmt.Println("Hello, BPF!")
}
```

I'll begin with a gccgo compilation, then do Go gc. (If you don't know the difference, try this summary by VonC: in short, gccgo can produce more optimized binaries, but for older versions of go.)

gccgo Function Counting

Compiling:

```
$ gccgo -o hello hello.go
$ ./hello
Hello, BPF!
```

Now I'll use my bcc tool funccount to dynamically trace and count all Go library functions that begin with "fmt.", while I reran the hello program in another terminal session:

```
# funccount 'go:fmt.*'
Tracing 160 functions for "go:fmt.*"... Hit Ctrl-C to end.
^C
FUNC                                    COUNT
fmt..import                                 1
fmt.padString.pN7_fmt.fmt                   1
fmt.fmt_s.pN7_fmt.fmt                       1
fmt.WriteString.pN10_fmt.buffer             1
fmt.free.pN6_fmt.pp                         1
fmt.fmtString.pN6_fmt.pp                    1
fmt.doPrint.pN6_fmt.pp                      1
fmt.init.pN7_fmt.fmt                        1
fmt.printArg.pN6_fmt.pp                     1
fmt.WriteByte.pN10_fmt.buffer               1
fmt.Println                                 1
fmt.truncate.pN7_fmt.fmt                    1
fmt.Fprintln                                1
fmt.$nested1                                1
fmt.newPrinter                              1
fmt.clearflags.pN7_fmt.fmt                  2
Detaching...
```

Neat! The output contains the fmt.Println() called by the program, along with other calls.

I didn't need to run Go under any special mode to do this, and I can walk up to an already running Go process and begin doing this instrumentation, without restarting it. So how does it even work?

- It uses Linux uprobes: User-Level Dynamic Tracing, added in Linux 3.5. It overwrites instructions with a soft interrupt to kernel instrumentation, and reverses the process when tracing has ended.
- The gccgo compiled output has a standard symbol table for function lookup.
- In this case, I'm instrumenting libgo (there's an assumed "lib" before this "go:"), as gccgo emits a dynamically linked binary. libgo has the fmt package.
- uprobes can attach to already running processes, or as I did here, instrument a binary and catch all processes that use it.
- For efficiency, I'm frequency counting the function calls in kernel context, and only emitting the counts to user space.

To the system, the binary looks like this:

```
$ file hello
hello: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib6
$ ls -lh hello
-rwxr-xr-x 1 bgregg root 29K Jan 12 21:18 hello
$ ldd hello
    linux-vdso.so.1 =>  (0x00007ffc4cb1a000)
    libgo.so.9 => /usr/lib/x86_64-linux-gnu/libgo.so.9 (0x00007f25f2407000)
    libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007f25f21f1000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f25f1e27000)
    /lib64/ld-linux-x86-64.so.2 (0x0000560b44960000)
    libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f25f1c0a000)
    libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f25f1901000)
$ objdump -tT /usr/lib/x86_64-linux-gnu/libgo.so.9 | grep fmt.Println
0000000001221070 g     O .data.rel.ro   0000000000000008              fmt.Println$descriptor
0000000000978090 g     F .text  0000000000000075              fmt.Println
0000000001221070 g    DO .data.rel.ro   0000000000000008  Base        fmt.Println$descriptor
0000000000978090 g    DF .text  0000000000000075  Base        fmt.Println
```

That looks a lot like a compiled C binary, which you can instrument using many existing debuggers and tracers, including bcc/BPF. It's a lot easier to instrument than runtimes that compile on the fly, like Java and Node.js. The only hitch so far is that function names can contain non-standard characters, like "." in this example.

funccount also has options like -p to match a PID, and -i to emit output every interval. It currently can only handle up to 1000 probes at a time, so "fmt.*" was ok, but matching everything in libgo:

```
# funccount 'go:*'
maximum of 1000 probes allowed, attempted 21178
```

... doesn't work yet. Like many things in bcc/BPF, when this limitation becomes too much of a nuisance we'll find a way to fix it.

Go gc Function Counting

Compiling using Go's gc compiler:

```
$ go build hello.go
$ ./hello
Hello, BPF!
```

Now counting the fmt functions:

```
# funccount '/home/bgregg/hello:fmt.*'
Tracing 78 functions for "/home/bgregg/hello:fmt.*"... Hit Ctrl-C to end.
^C
FUNC                                    COUNT
fmt.init.1                                  1
fmt.(*fmt).padString                        1
fmt.(*fmt).truncate                         1
fmt.(*fmt).fmt_s                            1
fmt.newPrinter                              1
fmt.(*pp).free                              1
fmt.Fprintln                                1
fmt.Println                                 1
fmt.(*pp).fmtString                         1
fmt.(*pp).printArg                          1
fmt.(*pp).doPrint                           1
fmt.glob.func1                              1
fmt.init                                    1
Detaching...
```

You can still trace fmt.Println(), but this is now finding it in the binary rather than libgo. Because:

```
$ file hello
hello: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, not stripped
$ ls -lh hello
-rwxr-xr-x 1 bgregg root 2.2M Jan 12 05:16 hello
$ ldd hello
    not a dynamic executable
$ objdump -t hello | grep fmt.Println
000000000045a680 g     F .text  00000000000000e0 fmt.Println
```

It's a 2 Mbyte static binary that contains the function.

Another difference is that the function names contain more unusual symbols: "*", "(", etc, which I suspect will trip up other debuggers until they are fixed to handle them (like bcc's trace was).

gccgo Function Tracing

Now I'll try Sasha Goldshtein's trace tool, also from bcc, to see per-event invocations of a function. Back using gccgo, and I'll start with this simple program from the go tour, functions.go:

```
package main

import "fmt"

func add(x int, y int) int {
    return x + y
}

func main() {
    fmt.Println(add(42, 13))
}
```

Now tracing the add() function:

```
# trace '/home/bgregg/functions:main.add'
PID    TID    COMM         FUNC
14424  14424  functions    main.add
```

... and with both its arguments:

```
# trace '/home/bgregg/functions:main.add "%d %d" arg1, arg2'
PID    TID    COMM         FUNC             -
14390  14390  functions    main.add         42 13
```

Awesome, that worked. Both arguments are printed on the right.

trace has other options (try -h), such as for including timestamps and stack traces with the output.

Go gc Function Tracing

Now the wheels start to go of the tracks... Same program, compiled with go build:

```
$ go build functions.go

# trace '/home/bgregg/functions:main.add "%d %d" arg1, arg2'
could not determine address of symbol main.add

$ objdump -t functions | grep main.add
$
```

No main.add()? Was it inlined? Disabling inlining:

```
$ go build -gcflags '-l' functions.go
$ objdump -t functions | grep main.add
0000000000401000 g     F .text  0000000000000020 main.add
```

Now it's back. Well that was easy. Tracing it and its arguments:

```
# trace '/home/bgregg/functions:main.add "%d %d" arg1, arg2'
PID    TID    COMM         FUNC             -
16061  16061  functions    main.add         536912504 16
```

That's wrong. The arguments should be 42 and 13, not 536912504 and 16.

Taking a peek with gdb:

```
$ gdb ./functions
[...]
warning: File "/usr/share/go-1.6/src/runtime/runtime-gdb.py" auto-loading has been declined
 by your `auto-load safe-path' set to "$debugdir:$datadir/auto-load".
[...]
(gdb) b main.add
Breakpoint 1 at 0x401000: file /home/bgregg/functions.go, line 6.
(gdb) r
Starting program: /home/bgregg/functions
[New LWP 16082]
[New LWP 16083]
[New LWP 16084]

Thread 1 "functions" hit Breakpoint 1, main.add (x=42, y=13, ~r2=4300314240) at
 /home/bgregg/functions.go:6
6           return x + y
(gdb) i r
rax            0xc820000180 859530330496
rbx            0x584ea0 5787296
rcx            0xc820000180 859530330496
rdx            0xc82005a048 859530698824
rsi            0x10 16
rdi            0xc82000a2a0 859530371744
rbp            0x0  0x0
rsp            0xc82003fed0 0xc82003fed0
r8             0x41 65
r9             0x41 65
r10            0x4d8ba0 5082016
r11            0x0  0
r12            0x10 16
r13            0x52a3c4 5415876
r14            0xa  10
r15            0x8  8
rip            0x401000 0x401000
eflags         0x206    [ PF IF ]
cs             0xe033   57395
ss             0xe02b   57387
ds             0x0  0
es             0x0  0
fs             0x0  0
gs             0x0  0
```

I included the startup warning about runtime-gdb.py, since it's helpful: if I want to dig deeper into Go context, I'll want to fix or source that. Even without it, gdb has shown the arguments as the variables "x=42, y=13".

I also dumped the registers to compare them to the x86_64 ABI, which is how bcc's trace reads them. From the syscall(2) man page:

```
       arch/ABI      arg1  arg2  arg3  arg4  arg5  arg6  arg7  Notes
       ──────────────────────────────────────────────────────────────
[...]
       x86_64        rdi   rsi   rdx   r10   r8    r9    -
```

42 and 13 don't appear rdi or rsi, or any of the registers. The reason is that Go's gc compiler is not following the standard AMD64 ABI function calling convention, which causes problems with this and other debuggers. This is pretty annoying. (I've also heard this complained about before, coincidentally, by my former colleagues). I guess Go needed to use a different ABI for return values, since it can return multiple values, so even if the entry arguments were standard we'd still run into differences.

I've browsed the Quick Guide to Go's Assembler and the Plan 9 assembly manual, and it looks like functions are passed on the stack. Here's our 42 and 13:

```
(gdb) x/3dg $rsp
0xc82003fed0:   4198477 42
0xc82003fee0:   13
```

BPF can dig these out too. As a proof of concept, I just hacked in a couple of new aliases, "go1" and "go2" for those entry arguments:

```
# trace '/home/bgregg/functions:main.add "%d %d" go1, go2'
PID    TID    COMM         FUNC             -
17555  17555  functions    main.add         42 13
```

Works. Hopefully by the time you read this post, I (or someone) has finished this work and added it to bcc trace tool. Preferably as "goarg1", "goarg2", etc.

Interface Arguments

I was going to trace the string argument to fmt.Println() as another example, but its argument is actually an "interface". From go's src/fmt/print.go:

```
func Println(a ...interface{}) (n int, err error) {
    return Fprintln(os.Stdout, a...)
```

With gdb you can dig out the string, eg, back to gccgo:

```
$ gdb ./hello
[...]
(gdb) b fmt.Println
Breakpoint 1 at 0x401c50
(gdb) r
Starting program: /home/bgregg/hello
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[New Thread 0x7ffff449c700 (LWP 16836)]
[New Thread 0x7ffff3098700 (LWP 16837)]
[Switching to Thread 0x7ffff3098700 (LWP 16837)]

Thread 3 "hello" hit Breakpoint 1, fmt.Println (a=...) at ../../../src/libgo/go/fmt/print.go:263
263 ../../../src/libgo/go/fmt/print.go: No such file or directory.
(gdb) p a
$1 = {__values = 0xc208000240, __count = 1, __capacity = 1}
(gdb) p a.__values
$18 = (struct {...} *) 0xc208000240
(gdb) p a.__values[0]
$20 = {__type_descriptor = 0x4037c0 <__go_tdn_string>, __object = 0xc208000210}
(gdb) x/s *0xc208000210
0x403483:   "Hello, BPF!"
```

So it can be read (and I'm sure there's an easier way with gdb, too). You could write a custom bcc/BPF program to dig this out, and we can add more aliases to bcc's trace program to deal with interface arguments.

Function Latency

(Update) Here's a quick demo of function latency tracing:

```
# funclatency 'go:fmt.Println'
Tracing 1 functions for "go:fmt.Println"... Hit Ctrl-C to end.
^C

Function = fmt.Println [3041]
     nsecs               : count     distribution
         0 -> 1          : 0        |                                        |
         2 -> 3          : 0        |                                        |
         4 -> 7          : 0        |                                        |
         8 -> 15         : 0        |                                        |
        16 -> 31         : 0        |                                        |
        32 -> 63         : 0        |                                        |
        64 -> 127        : 0        |                                        |
       128 -> 255        : 0        |                                        |
       256 -> 511        : 0        |                                        |
       512 -> 1023       : 0        |                                        |
      1024 -> 2047       : 0        |                                        |
      2048 -> 4095       : 0        |                                        |
      4096 -> 8191       : 0        |                                        |
      8192 -> 16383      : 27       |****************************************|
     16384 -> 32767      : 3        |****                                    |
Detaching...
```

That's showing a histogram of latency (in nanoseconds) for fmt.Println(), which I was calling in a loop.

WARNING: There are some unfortunate problems with this: if Go switches a goroutine to a different OS thread during the function call, then funclatency won't match the entry to the return. We'll need a new tool, `gofunclatency`, that uses Go's internal GOID for latency tracking instead of the OS's TID. There may also be problems with uretprobes modifying Go in a way that causes Go to crash, which we'll need to debug and figure out a plan around. See the comment by Suresh for details.

Next Steps

I took a quick look at Golang with dynamic tracing and Linux enhanced BPF, via bcc's funccount and trace tools, with some successes and some challenges. Counting function calls works already. Tracing function arguments when compiled with gccgo also works, whereas Go's gc compiler doesn't follow the standard ABI calling convention, so the tools need to be updated to support this. As a proof of concept I modified the bcc trace tool to show it could be done, but that feature needs to be coded properly and integrated. Processing interface objects will also be a challenge, and multi-return values, again, areas where we can improve the tools to make this easier, as well as add macros to C for writing other custom Go observability tools as well.

Hopefully there will be a follow up post (not necessarily by me, feel free to take up the baton if this interests you) that shows improvements to bcc/BPF Go gc argument tracing, interfaces, and return values.

Another important tracing topic, which again can be a follow up post, is stack traces. Thankfully Go made frame pointer-based stack traces default in 1.7.

Lastly, another important topic that could be a post by itself is tracing Go functions along with kernel context. BPF and bcc can instrument kernel functions, as well as user space, and I can imagine custom new tools that combine information from both.

You can comment here, but I can't guarantee your comment will remain here forever: I might switch comment systems at some point (eg, if disqus add advertisements).