Evaluation of tail call costs in eBPF

Clément Joly, François Serman
Agenda

- Problem
- Benchmark 1
- Benchmark 2
- Future
Problem
Cloudflare

- DDoS mitigation on all machines, in our 200+ PoPs
- L4Drop: XDP / eBPF
  - Set of rules (Rule sets)
  - Tail calls

https://blog.cloudflare.com/cloudflare-architecture-and-how-bpf-eats-the-world/
Tail calls

C code

tail_call((void *)ctxt, &MAP, 0);
Improved tail calls (Daniel Borkmann)

- Avoid the retpoline overhead
  - Direct jump with static map
- Apply to L4Drop
- V5.4 vs v5.5

https://lwn.net/ml/netdev/cover.1574452833.git.daniel@iogearbox.net/
Tail calls

C code

tail_call((void *)ctxt, &MAP, 0);

eBPF assembly

0: (b7) r3 = 0
1: (18) r2 = map[id:526]
3: (85) call bpf_tail_call#12
4: (b7) r0 = 1

From https://cilium.io/blog/2020/02/18/cilium-17/
x86-64 eBPF JITed: Before

```
19:    xor %edx,%edx         |_ Index (r3 = 0)
1b:    movabs $0xfffff88d95cc82600,%rsi    |_ map (r2 = map[id:526])
25:    mov %edx,%edx         | index >= array->map.max_entries
27:    cmp %edx,0x24(%rsi)    | check
2a:    jbe 0x0000000000000066 | |
2c:    mov -0x224(%rbp),%eax | tail call limit check
32:    cmp $0x20,%eax        | |
35:    ja 0x0000000000000066 | |
37:    add $0x1,%eax         | |
3a:    mov %eax,-0x224(%rbp) | |
40:    mov 0xd8(%rsl,%rdx,8),%rax | prog = array->ptrs[index]
48:    test %rax,%rax        | prog == NULL check
4b:    je 0x0000000000000066 | |
4d:    mov 0x30(%rax),%rax   | goto *(prog->bpf_func + prologue_size)
51:    add $0x19,%rax        | |
55:    callq 0x0000000000000061 | retoline for indirect jump
5a:    pause                | |
5c:    lfence               | |
5f:    jmp 0x000000000000005a | |
61:    mov %rax,(%rsp)       | |
65:    retq                 | |
```

From [https://cilium.io/blog/2020/02/18/cilium-17/](https://cilium.io/blog/2020/02/18/cilium-17/)
x86-64 eBPF JITed: After (Daniel Borkmann)

19:    xor %edx,%edx  | _ index (r3 = 0)
1b:    movabs $0xffff9d8afd74c000,%rsi  | _ map (r2 = map[id:526])
25:    mov -0x224(%rbp),%eax  | tail call limit check
2b:    cmp $0x20,%eax  |
2e:    ja 0x000000000000003e  |
30:    add $0x1,%eax  |
33:    mov %eax,-0x224(%rbp)  |
39:    jmpq 0xffffffffffffd1785  | _ [direct] goto *(prog->bpf_func + prologue_size)
3e:    mov $0x1,%eax  |

(next instruction, r0 = 1)

From https://cilium.io/blog/2020/02/18/cilium-17/
Measuring

- No existing, well-known solution specific enough
Benchmark 1
BPF_PROG_TEST_RUN

- One packet
- One XDP program
- Multiple runs
  - Returns average time

https://lwn.net/Articles/718784/
Benchmark 1

- Advantage: stable, one program
- Drawback: not production-like
# Machine: 9th generation

<table>
<thead>
<tr>
<th>Machine name</th>
<th>CPU</th>
<th>Number of core (logical)</th>
<th>Frequency (GHz)</th>
<th>Maximum frequency (GHz)</th>
<th>RAM (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>testM8</td>
<td>Intel(R) Xeon(R) Platinum 6162</td>
<td>96</td>
<td>1.90</td>
<td>3.50</td>
<td>188</td>
</tr>
</tbody>
</table>

[https://blog.cloudflare.com/a-tour-inside-cloudflares-g9-servers/](https://blog.cloudflare.com/a-tour-inside-cloudflares-g9-servers/)
Measuring tail call CPU time cost

- **Single tail call**
  - Difficult to get the diff

- **20 rulesets of one rule**
  - With tail calls
  - Merged

- **Same number of instructions**
  - Difference -> cost of tail calls
Benchmark 1

- Average cost per tail call
  - 25 -> 5 ns
Benchmark 2
2 network namespaces

- Deploy L4Drop + load balancer & sampler
  - Kernel probes & eBPF
- Simulate traffic
  - `ip netns exec m1 iperf3 -c <ip_srv> -N -J`
KProbes

- Entry time, exit time
  - Stored in a map
- Bias due to code around
- Here, on veth_poll
Sample data

| @less_16: count 1933452, average 1742, total 3369543330 |
| @more_16: count 965732, average 61751, total 59635349493 |
| @ns: |
| [512, 1K) | 1 |
| [1K, 2K) | 1512012 |
| [2K, 4K) | 410325 |
| [4K, 8K) | 9240 |
| [8K, 16K) | 1890 |
| [16K, 32K) | 668 |
| [32K, 64K) | 569424 |
| [64K, 128K) | 394983 |
| [128K, 256K) | 499 |
| [256K, 512K) | 78 |
| [512K, 1M) | 64 |
| @stats: count 2899184, average 21731, total 63004892823 |
## Sample data

| less_16: count 1933452, average 1742, total 3369543330 |
| more_16: count 965732, average 61751, total 59635349493 |

### < 16000ns

- [512, 1K) 1
- [1K, 2K) 1512012
- [2K, 4K) 410325
- [4K, 8K) 9246
- [8K, 16K) 1890

### > 16000ns

- [16K, 32K) 668
- [32K, 64K) 569424
- [64K, 128K) 394983
- [128K, 256K) 499
- [256K, 512K) 78
- [512K, 1M) 64

@stats: count 2899184, average 21731, total 63004892823
Benchmark 2

- Benefit: production-like conditions
- Drawback: complexity, less precision
  - Surrounding code
Benchmark 2 (CPU)

- Roughly 20% gain
  - Due to code in the kprobed function
- Graphs on global average
Benchmark 2 (throughput)

- Slower eBPF program → lower throughput
- 17-23 % gain
- Less reliable
Future
Drawbacks

- Clusters of values with KProbes
- Overhead with IPerf between namespaces
  - Not always precise enough
- More direct measurements for specific instructions (tail calls for instance)?
Perf

- Difficult to isolate various eBPF programs
  - Not isolated either in the second benchmark
- Not used at first
  - Record: not enough to distinguish
  - Trace: too much overhead
Bpftool prog profile
Bpf stats enabled

run_time_ns / run_cnt

bash-5.0# sysctl -w kernel.bpf_stats_enabled=1
kernel.bpf_stats_enabled = 1
bash-5.0# bpftool prog show id 1

1: xdp  tag 3b185187f1855c4c  gpl run_time_ns 20881  run_cnt 148
loaded_at 2020-08-17T22:01:04+0200  uid 0
xlated 16B  jited 35B  memlock 4096B
bash-5.0#
Conclusion
Thanks