Performance Analysis of XDP Programs

LISA 21

Zachary H. Jones, Verizon Media Platform

Verizon Media Platform

130 Tbps
Network Capacity

168+
PoPs

6
Continents

7K+
Interconnects

North America

PoPs
Ashburn
Atlanta
Boston
Chicago
Dallas
Denver
Detroit
Guadalajara
Houston
Las Vegas
Los Angeles
Mexico City
Miami
New York
Philadelphia
Phoenix
Pittsburgh
Puebla
Querétaro
San Jose
Seattle
Washington D.C.

South America

PoPs
Barranquilla
Bogota
Buenos Aires
Lima
Medellín
Rio de Janeiro
Santiago
São Paulo
Valparaíso

Europe

PoPs
Amsterdam
Bucharest
Copenhagen
Frankfurt
Helsinki
Lisbon
London
Madrid
Manchester
Marseille
Milan
Munich
Paris
Riga
Sofia
Stockholm
Vienna
Warsaw

Middle East

PoPs
Fujairah
Kuwait
Muscat

Africa

PoPs
Johannesburg
Nairobi

Asia

PoPs
Bangalore
Bangkok
Chennai
Hong Kong
Jakarta
Kaohsiung
Manila
Mumbai
New Delhi
Osaka
Seoul
Singapore
Taipei
Tokyo

Oceania

PoPs
Auckland
Melbourne
Sydney

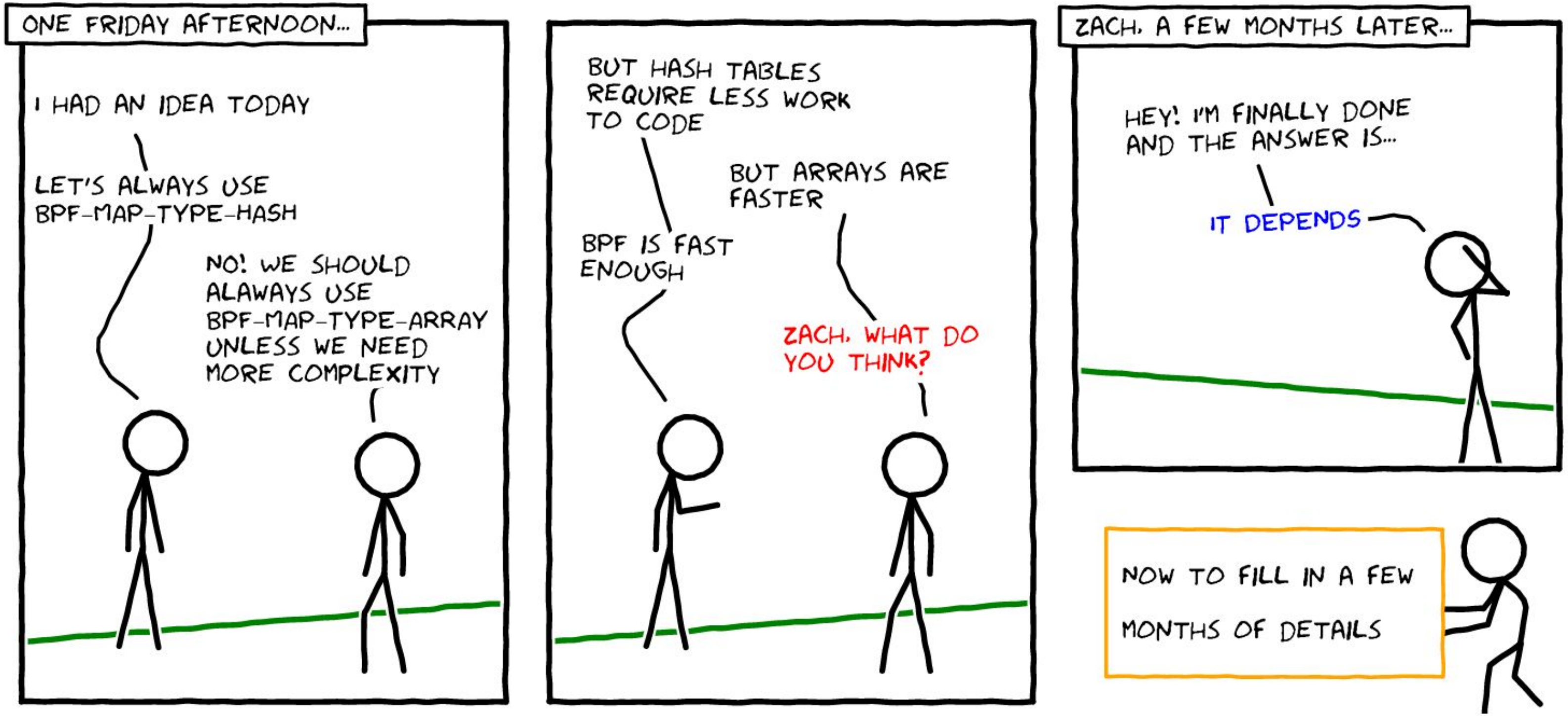
What is XDP (and BPF)?

- **XDP (eXpress Data Path) is a BPF based *high performance* packet processor in the Linux networking stack[1]**
  - It does not require specialized hardware, bypass the kernel, or replace the TCP/IP stack
  - For higher performance, it does require driver support
- **BPF (Berkeley Packet Filter) is a virtual machine-like construct inside the Linux kernel to allow *safe execution* of arbitrary bytecode[2]**

- **More information:**
  - https://www.iovisor.org/technology/xdp
  - https://ebpf.io/
  - https://github.com/iovisor/bpf-docs/blob/master/Express_Data_Path.pdf

verizon media

How did I get here?



Made with http://cmx.io

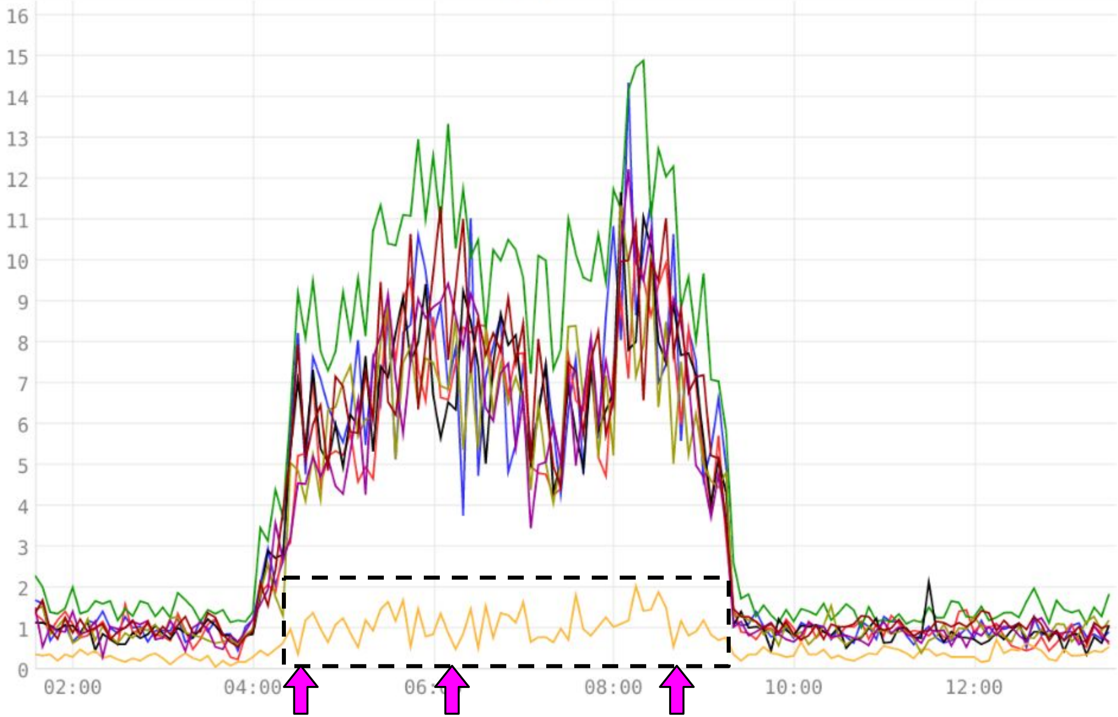
A few months of details...

1. **Why is measuring XDP/BPF performance hard?**
2. Our approach for XDP performance analysis
3. Outcomes
4. ~~Future~~ Ongoing Work

verizon media

Observing XDP Test Deployment

- Ran a XDP program on one server in a production test POP
- Increased load in the POP
- Observed difference in SoftIRQ cpu utilization between traditional kernel code and XDP acceleration code
- XDP test server (orange line on the bottom) utilization is much lower
- Is it too low?



Soft IRQ CPU Utilization

verizon media

Think about what we are observing

- **I could not explain the CPU utilization behavior.**

- **Brendan Gregg summarizes the concept of Active Benchmarking**
  - “With active benchmarking, you analyze performance while the benchmark is still running (not just after it's done), using other tools. You can confirm that the benchmark tests what you intend it to, and that you understand what that is.”[3]

- **I went back to our test lab and investigated further**
  - Synthetic load against an XDP program at 20Mpps (64 byte packets)
  - System under test configured using 20 RSS CPUs
  - 1Mpps per RSS CPU was consuming 1% soft IRQ utilization on each RSS CPU

Think about what we are observing (cont'd)

- **1% of a CPU is 10ms**
  - 10ms for 1M pkts is 10ns per packet, 10ns on a 3GHz is 30 cycles per packet
- **30 cycles to run driver and XDP code**
  - Parse IP address, port, TCP flags, window length, ttl, lookup connection in state table, expand packet buffer to add VLAN tag, rewrite packet mac address, go through driver Tx XDP code
- **That is 3 cycles per task.**
  - It takes approximately 2 to 10 cycles for L1 and L2 cache data accesses alone
- **Linux Network code, interrupt handler, Rx cleanup path all need to run in the 30 cycles.**
- ***The utilization is unrealistically low.***
- **Before this point, I was doing what Gregg calls "casual benchmarking"**
  - "You benchmark A, but actually measure B, and conclude you've measured C."[3]

verizon media

Time Accounting is a Hard Problem

- **CPU utilization measurements in utilities like top and sysstat are skewed.**

- **Linux has multiple ways to keep track of time**
  - See `CONFIG_*_ACCOUNTING_*` kernel options
  - Different Linux distros use different default accounting methods
  - The accounting methods impact the kernel's perception of time

- **XDP code executes in interrupt driven soft IRQ context**

Time Accounting is a Hard Problem (cont'd)

- **Timekeeping precision in hard and soft IRQs contextes is challenging**
  - Accounting based on ticks can miss entire softirq periods.
  - Accounting based on hard/soft IRQ transitions is more accurate (can still be skewed by 8%[4]) but adds overhead to every transition.
  - Idle polling (C-state 0 only) can help mitigate inaccuracies of tick accounting

- **Hardware performance counters can be used for CPU utilization reporting as well**

This is not a talk about CPU utilization…

- **…but that 1% was really 60%**
  - CPU utilization measurements was good reinforcement for Active Benchmarking
  - XDP/BPF is fast…but not 30 cycles per packet fast for our workload
- **The performance of XDP/BPF does come at a cost**
  - BPF code is jitted and inlined aggressively
    - Call graphs are flat, no bpf to bpf function calls before 4.16
  - XDP programs exist within the limits of New API(NAPI) networking interface
    - Scaling limited by overhead of interrupts, time slicing, DMA management, etc
  - BPF programs do not have kprobe hooks
    - With kernel 5.5 or newer and `CONFIG_DEBUG_BTF=y`, you can attach fentry/fexit BPF prog probes and profile BPF progs
- **BPF is moving fast and features are being introduced rapidly**

A few months of details...

1. Why is measuring XDP/BPF performance hard?
2. **An approach for XDP performance analysis**
3. Outcomes
4. Ongoing Work

verizon media

Focus is on Performance of BPF code

- **This is not a talk about arrays vs hash tables...**
  - ...we already know the answer is “it depends”
  - But, the conversation was a “springboard” to get serious about performance analysis
- **Will not cover XDP-adjacent tuning topics**
  - RSS, RPS
  - netdev budgets
  - CPU affinity, isolation
  - RCU tuning
  - cpupower
- **Will mention XDP-adjacent tuning efforts at times**

verizon media

Three Components to Our Approach

Analytical Budget Time Measurement

Goal: To understand where CPU time is being spent and remaining free time

Building Block Microbenchmarks

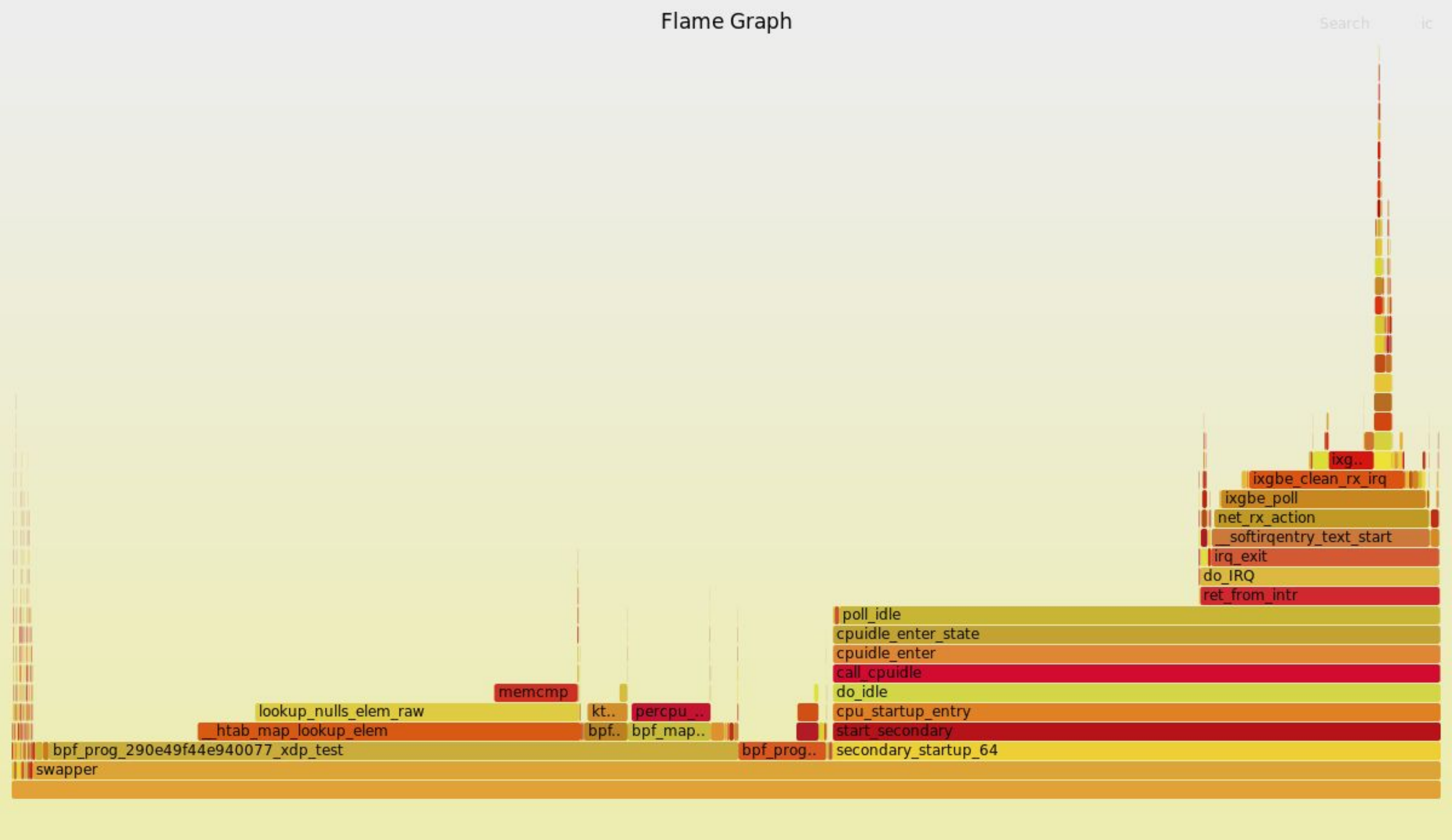
Goal: To have an understanding of pathlength cost of BPF helpers and data structures in ideal situations

Instruction level Sampling and Annotation

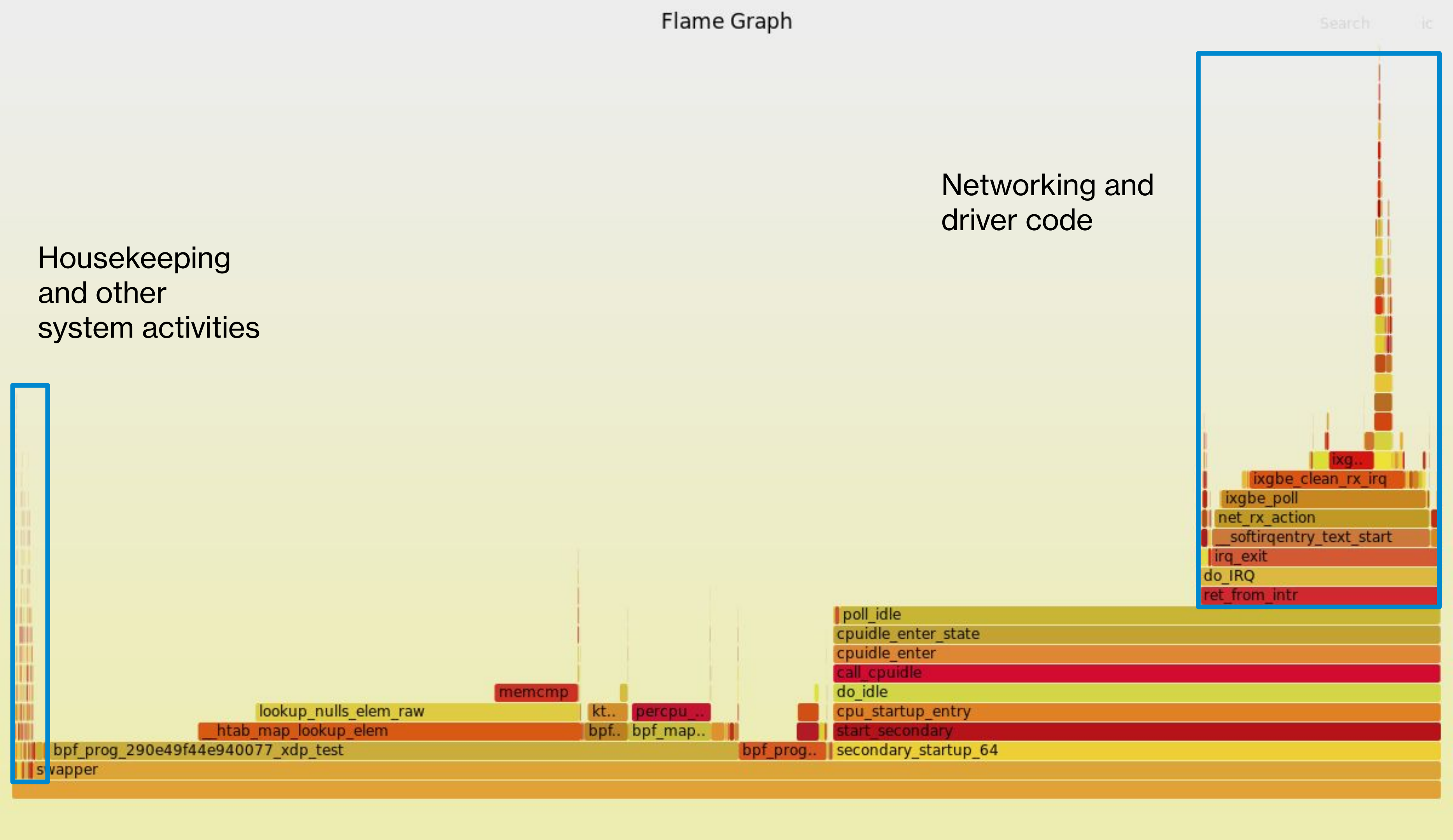
Goal: To be able to find hot spots and common branches in our BPF code

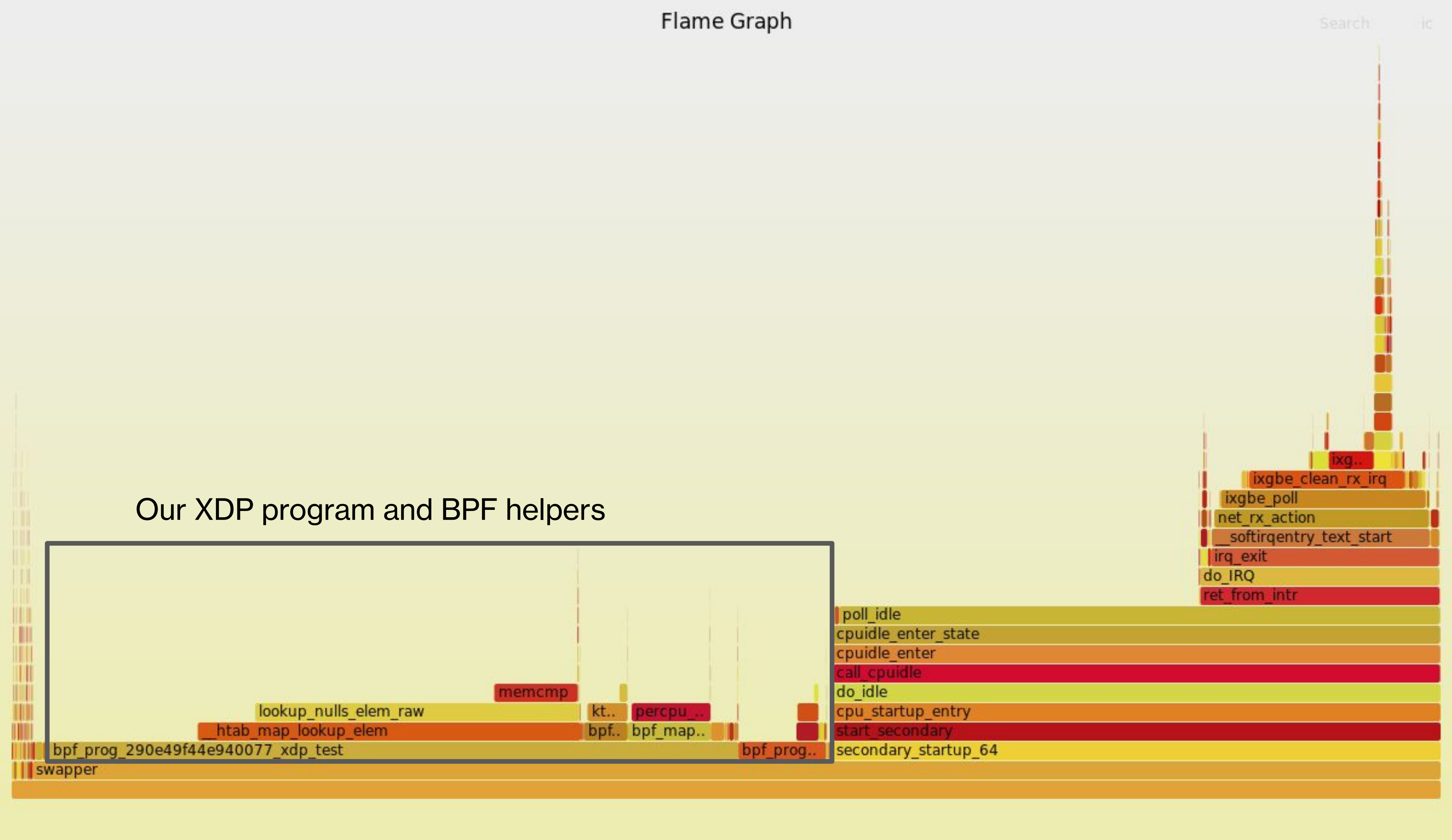
Analytical budget time measurement

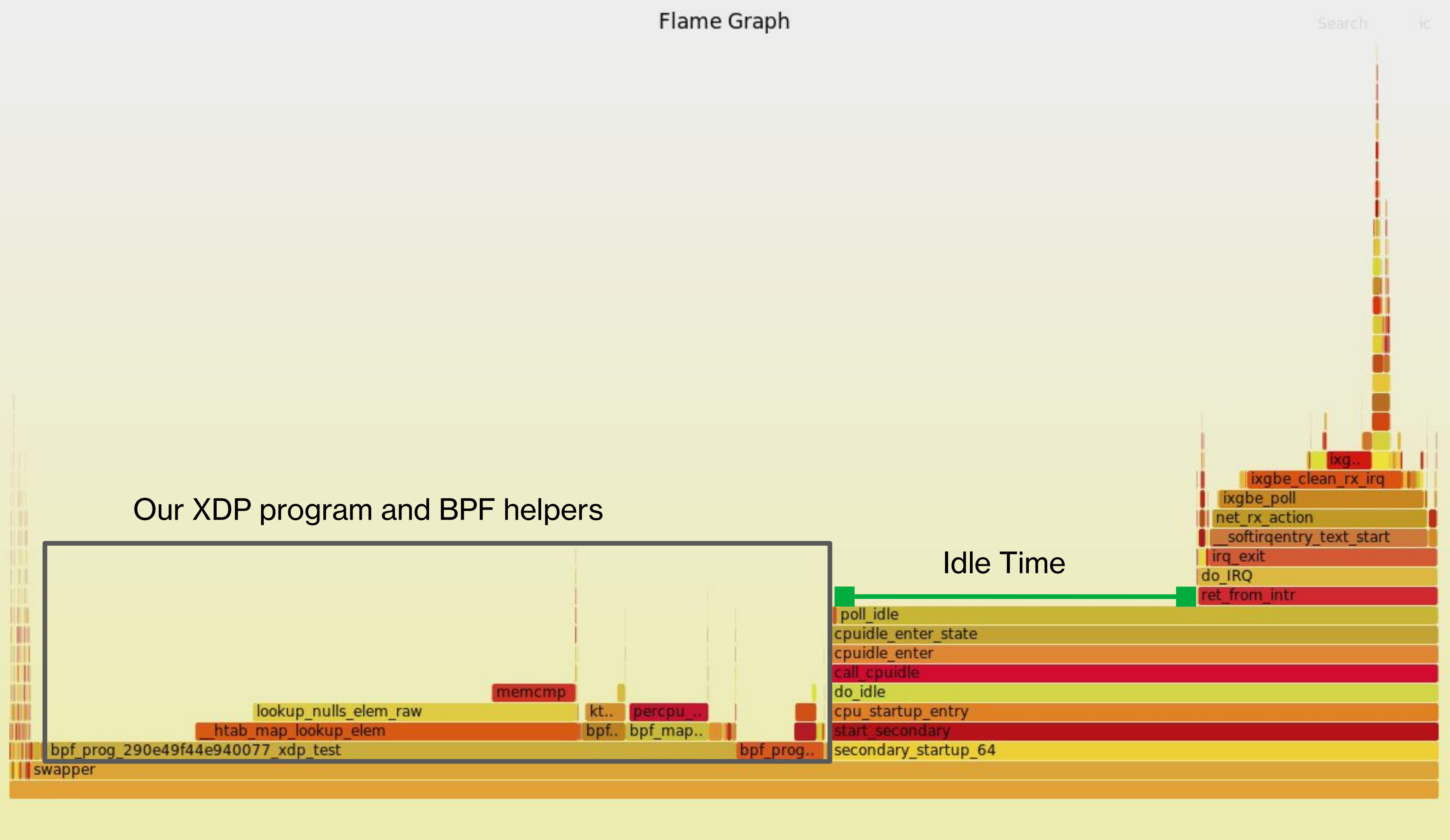
- **Use flame graphs to visualize what the CPU is doing**
  - http://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html
  - Can see spires for XDP code, networking, and other code running on the CPU
  - Networking code does not scale linear with load, so plan tests accordingly
- **Flame graphs will not show idle time because the CPU is asleep without tuning the kernel**
  - Add "`idle=poll`" to GRUB kernel arg line
  - Use `cpupower` to disable all C-states dynamically
  - *Note: Using "`intel_idle.max_cstate=0`" disables the intel_idle driver but under certain circumstances will leave only C-state 1 enabled*
- **Record performance data for single CPU**
  - Configured P-states to run all cores at All-Core Turbo speed to ensure consistency
  - Record one of the RSS CPUs running XDP program (`perf record -c <cpu num>`)



Flame Graph
Search
ic
ixg..
ixgbe_clean_rx_irq
ixgbe_poll
net_rx_action
__softirqentry_text_start
irq_exit
do_IRQ
ret_from_intr
poll_idle
cpuidle_enter_state
cpuidle_enter
call_cpuidle
memcmp
do_idle
lookup_nulls_elem_raw
kt..
percpu_..
cpu_startup_entry
__htab_map_lookup_elem
bpf..
bpf_map..
start_secondary
bpf_prog_290e49f44e940077_xdp_test
bpf_prog..
secondary_startup_64
swapper

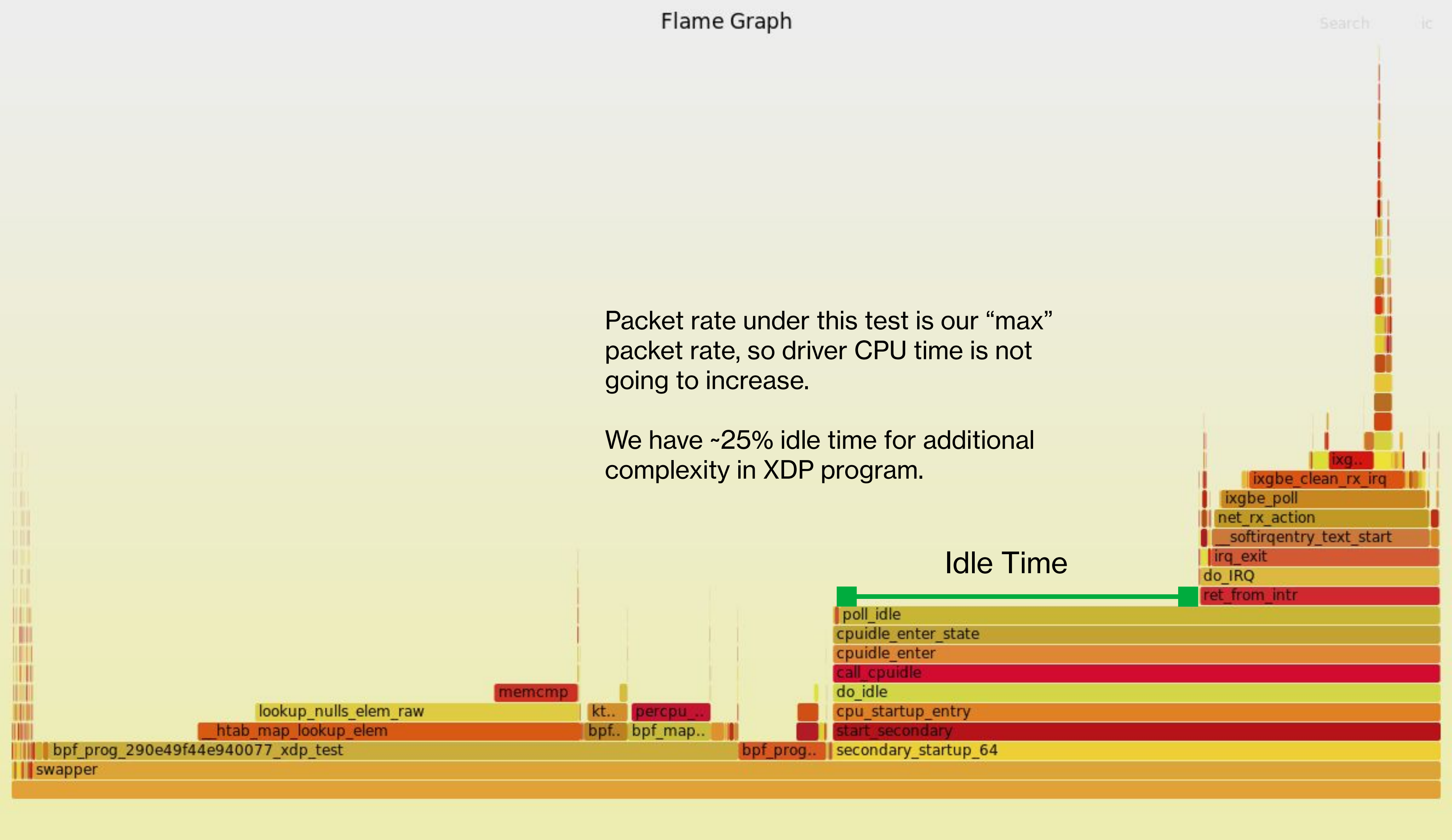

Flame Graph
Search
ic
Networking and driver code
Housekeeping and other system activities
ixg..
ixgbe_clean_rx_irq
ixgbe_poll
net_rx_action
__softirqentry_text_start
irq_exit
do_IRQ
ret_from_intr
poll_idle
cpuidle_enter_state
cpuidle_enter
call_cpuidle
do_idle
cpu_startup_entry
start_secondary
secondary_startup_64
memcmp
lookup_nulls_elem_raw
kt..
percpu_..
__htab_map_lookup_elem
bpf..
bpf_map..
bpf_prog_290e49f44e940077_xdp_test
bpf_prog..
swapper


Flame Graph
Search
Our XDP program and BPF helpers
memcmp
lookup_nulls_elem_raw
kt..
percpu_..
__htab_map_lookup_elem
bpf..
bpf_map..
bpf_prog_290e49f44e940077_xdp_test
bpf_prog..
swapper
ixg..
ixgbe_clean_rx_irq
ixgbe_poll
net_rx_action
__softirqentry_text_start
irq_exit
do_IRQ
ret_from_intr
poll_idle
cpuidle_enter_state
cpuidle_enter
call_cpuidle
do_idle
cpu_startup_entry
start_secondary
secondary_startup_64


Flame Graph
Search
ic
Our XDP program and BPF helpers
Idle Time
ixg..
ixgbe_clean_rx_irq
ixgbe_poll
net_rx_action
__softirqentry_text_start
irq_exit
do_IRQ
ret_from_intr
memcmp
lookup_nulls_elem_raw
kt..
percpu_..
__htab_map_lookup_elem
bpf..
bpf_map..
bpf_prog_290e49f44e940077_xdp_test
bpf_prog..
poll_idle
cpuidle_enter_state
cpuidle_enter
call_cpuidle
do_idle
cpu_startup_entry
start_secondary
secondary_startup_64
swapper

Packet rate under this test is our “max” packet rate, so driver CPU time is not going to increase.

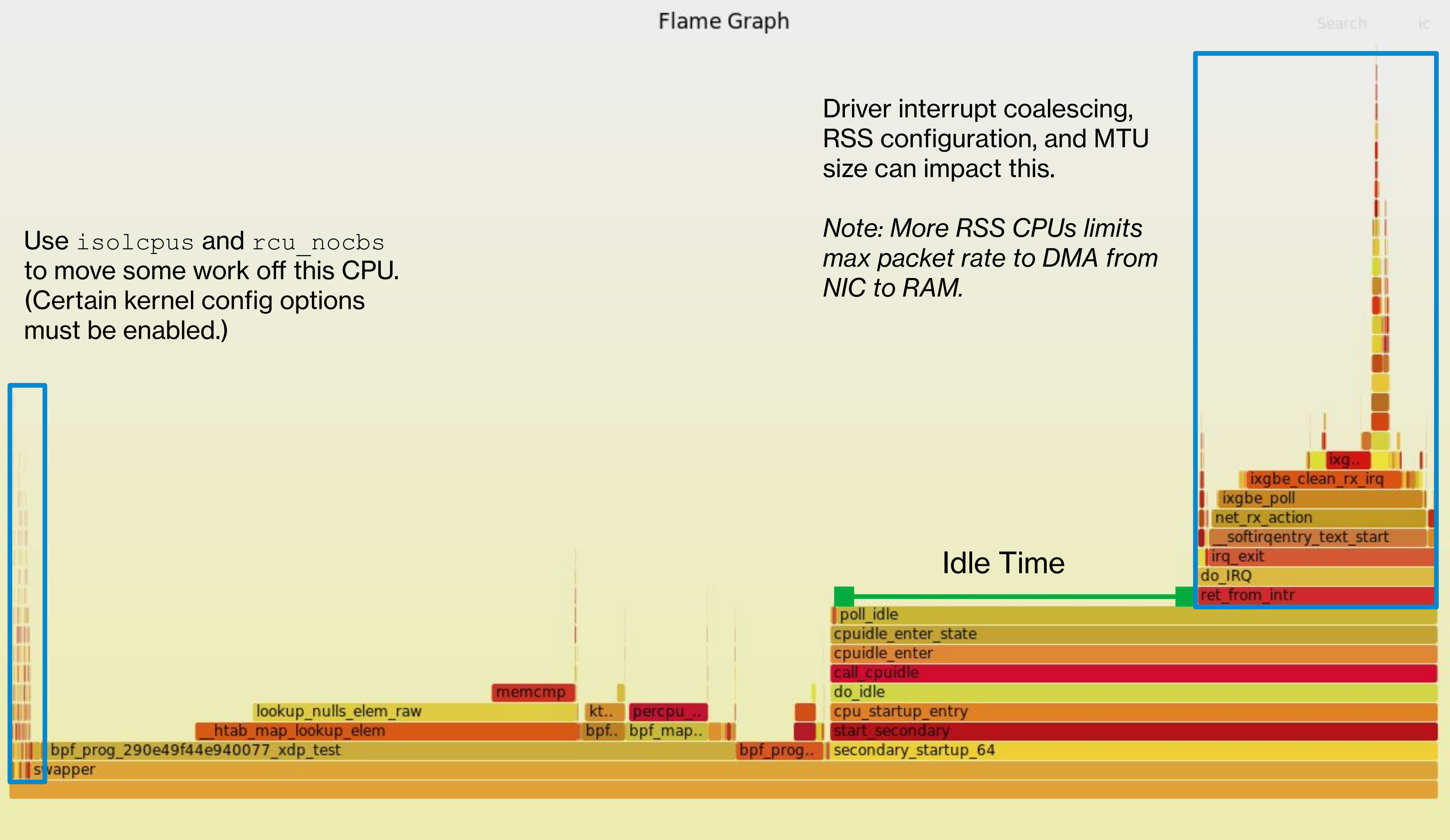
We have ~25% idle time for additional complexity in XDP program.



Use `isolcpus` and `rcu_nocbs` to move some work off this CPU. (Certain kernel config options must be enabled.)

Driver interrupt coalescing, RSS configuration, and MTU size can impact this.

Note: More RSS CPUs limits max packet rate to DMA from NIC to RAM.



Microbenchmarking Building Blocks

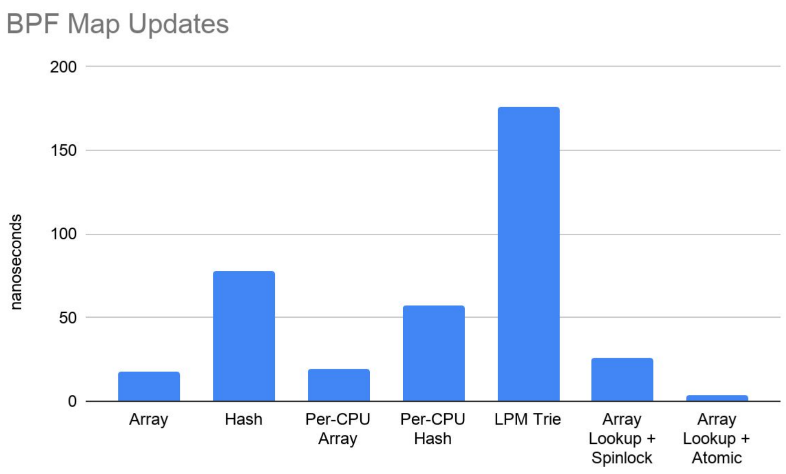
- **Kernel provides BPF_PROG_TEST_RUN command for the bpf syscall to run N iterations of your BPF program and using the kernel BPF time keeping to return average runtime for the N iterations.**
  - Example:

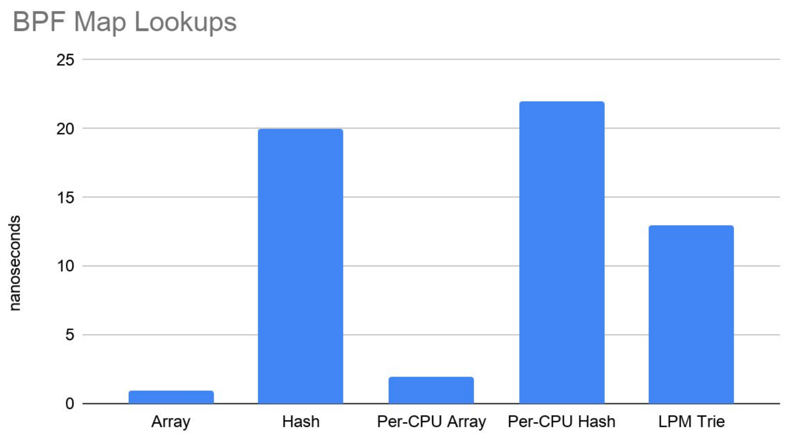
    ```
    >$ bpftool prog loadall ./xdp_test.o /sys/fs/bpf/my_xdp_test
    >$ bpftool prog run pinned /sys/fs/bpf/my_xdp_test/xdp-test data_in pkt.in repeat 1000000
    Return value: 0, duration (average): 10ns
    ```

  - An alternative path is to build a test harness program using `bpf_prog_load()` and `bpf_prog_test_run_xattr()`
- **To measure the performance of a component or helper (e.g.** `bpf_get_numa_node_id()`**)**
  - Perform a test run of N iterations of a pass through program (return XDP_PASS)
  - Perform a test run of N iterations of a program that uses the component or helper
  - Measure the difference in average runtime between the two programs

verizon media

So...Arrays or Hash Tables?

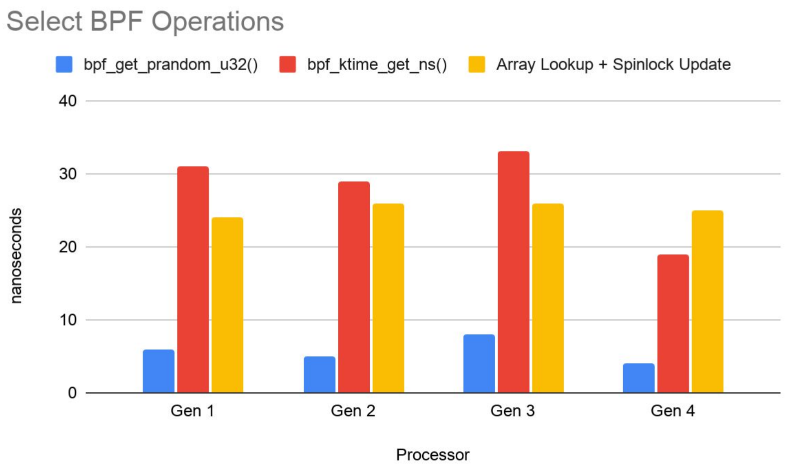


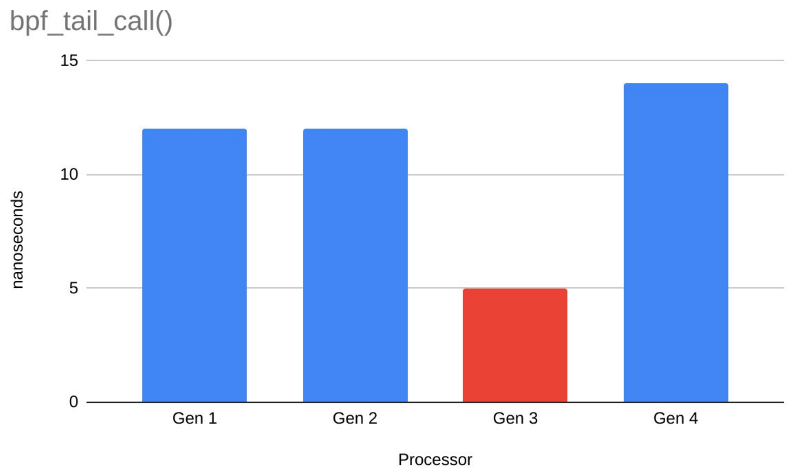


- The benchmark is a single consumer/producer model, so normal verses per-CPU comparisons do not take into account contention from multiple producers or consumers.
- The benchmark just shows pathlength of the operations
- The answer is still “it depends”

verizon media

CPU Architecture Comparison



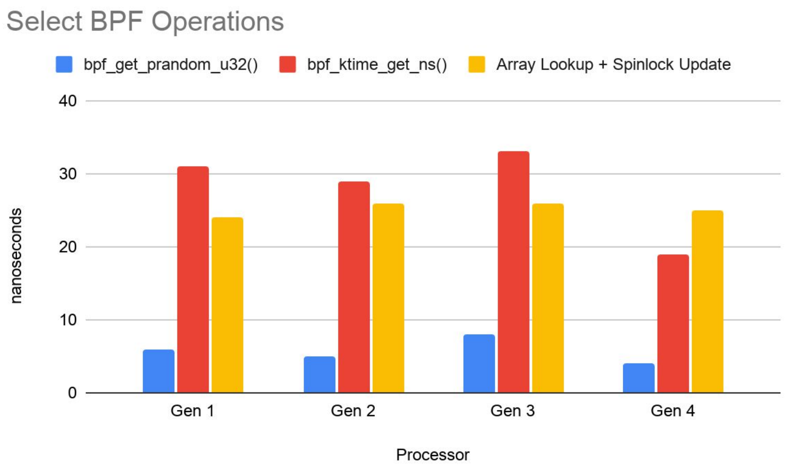


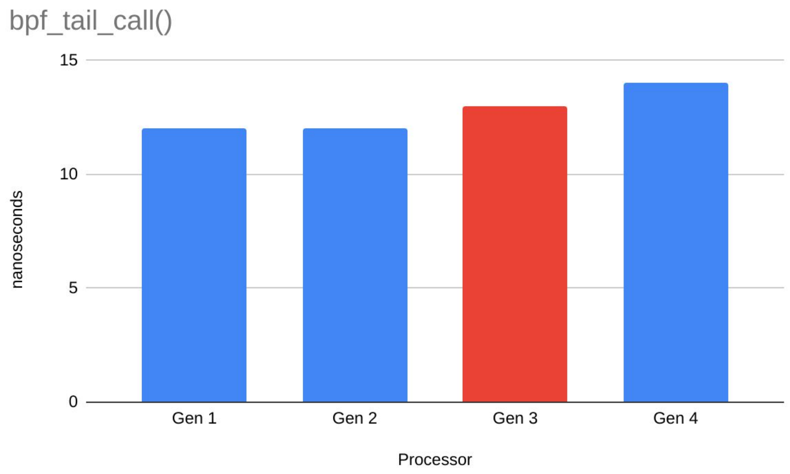
What makes System 3 unique?

This was a result of a configuration difference. Gen 3 system had Spectre/Meltdown mitigations disabled.

verizon media

CPU Architecture Comparison





verizon media

Instruction level sampling and annotation

- **Use annotation feature of `perf` to view annotated assembly code with utilization percentages**
  - The profiler looks at the x86_64 assembly code output from the BPF JIT
  - BTF annotations show BPF C code inline
- **Prerequisites**
  - `perf` from kernel >= 5.1
  - `perf` linked against libopcode (binutils-dev[el]) during compilation
  - Compile BPF programs using clang >= 10
- **Use**
  - Use `perf record` with your favorite arguments
  - Load profile data with `perf annotate` or `perf report`
    - When in a report, highlight the function of interest press 'a'

verizon media

```
Samples: 1K of event 'cycles', 4000 Hz, Event count (approx.): 333611469
bpf_prog_54ee63da2d576a5e_xdp_test  bpf_prog_54ee63da2d576a5e_xdp_test [P
Percent│     int xdp_test(struct xdp_md *ctx) {
       │       nop
       │       xchg   %ax,%ax
       │       push   %rbp
  4.00 │       mov    %rsp,%rbp
       │       sub    $0x8,%rsp
       │       mov    $0x7b,%edi
       │       __u32 key = TEST_U32_KEY;
 36.00 │       mov    %edi,-0x4(%rbp)
       │       mov    %rbp,%rsi
       │
       │       add    $0xfffffffffffffffc,%rsi
       │       __u64 *v = bpf_map_lookup_elem(&test_map, &key);
       │       movabs $0xffff917fb0ad4000,%rdi
  8.00 │       callq  *ffffffffe32ba060
       │       mov    %rax,%rdi
       │       if(v && *v == TEST_U64_VAL)
       │       test   %rdi,%rdi
       │       je     4a
       │       mov    $0x2,%eax
       │       if(v && *v == TEST_U64_VAL)
 24.00 │       mov    0x0(%rdi),%rdi
       │       if(v && *v == TEST_U64_VAL)
       │       cmp    $0x499602d2,%rdi
       │       je     4c
       │4a:    xor    %eax,%eax
       │     }
 28.00 │4c:    leaveq
       │       retq
```

```
Samples: 1K of event 'cycles', 4000 Hz, Event count (approx.): 323918153
bpf_prog_5f78d0a2aee6f57a_xdp_test  bpf_prog_5f78d0a2aee6f57a_xdp_test [P
Percent│     int xdp_test(struct xdp_md *ctx) {
       │       nop
       │       xchg   %ax,%ax
       │       push   %rbp
  5.41 │       mov    %rsp,%rbp
       │       sub    $0x8,%rsp
       │       mov    $0xffffffff,%edi
       │       __u32 key = -1;
 40.54 │       mov    %edi,-0x4(%rbp)
       │       mov    %rbp,%rsi
       │
       │       add    $0xfffffffffffffffc,%rsi
       │       __u64 *v = bpf_map_lookup_elem(&test_map, &key);
       │       movabs $0xffff917fb02f4000,%rdi
 17.57 │       callq  *ffffffffe32c0324
       │       mov    %rax,%rdi
       │       if(v && *v == TEST_U64_VAL)
       │       test   %rdi,%rdi
 21.62 │       je     4a
       │       mov    $0x2,%eax
       │       if(v && *v == TEST_U64_VAL)
       │       mov    0x0(%rdi),%rdi
       │       if(v && *v == TEST_U64_VAL)
       │       cmp    $0x499602d2,%rdi
       │       je     4c
       │4a:    xor    %eax,%eax
       │     }
 14.87 │4c:    leaveq
       │       retq
```

- **Same trivial function profiled on left and right, but with one difference**
- Look up element in array and compare to value.
- Left side looks up valid index; right side looks up invalid index; annotations show the branch difference

```
Samples: 1K of event 'cycles', 4000 Hz, Event count (approx.): 333611469
bpf_prog_54ee63da2d576a5e_xdp_test  bpf_prog_54ee63da2d576a5e_xdp_test [P
Percent │      int xdp_test(struct xdp_md *ctx) {
        │        nop
        │        xchg   %ax,%ax
        │        push   %rbp
   4.00 │        mov    %rsp,%rbp
        │        sub    $0x8,%rsp
        │        mov    $0x7b,%edi
        │        __u32 key = TEST_U32_KEY;
  36.00 │        mov    %edi,-0x4(%rbp)
        │        mov    %rbp,%rsi
        │
        │        add    $0xfffffffffffffffc,%rsi
        │ ➡      __u64 *v = bpf_map_lookup_elem(&test_map, &key);
        │        movabs $0xffff917fb0ad4000,%rdi
   8.00 │        callq  *ffffffffe32ba060
        │        mov    %rax,%rdi
        │ ➡      if(v && *v == TEST_U64_VAL)
        │        test   %rdi,%rdi
        │        je     4a
        │        mov    $0x2,%eax
        │ ➡      if(v && *v == TEST_U64_VAL)
  24.00 │        mov    0x0(%rdi),%rdi
        │ ➡      if(v && *v == TEST_U64_VAL)
        │        cmp    $0x499602d2,%rdi
        │        je     4c
        │ 4a:    xor    %eax,%eax
        │      }
  28.00 │ 4c:    leaveq
        │        retq
```

```
Samples: 1K of event 'cycles', 4000 Hz, Event count (approx.): 323918153
bpf_prog_5f78d0a2aee6f57a_xdp_test  bpf_prog_5f78d0a2aee6f57a_xdp_test [P
Percent │      int xdp_test(struct xdp_md *ctx) {
        │        nop
        │        xchg   %ax,%ax
        │        push   %rbp
   5.41 │        mov    %rsp,%rbp
        │        sub    $0x8,%rsp
        │        mov    $0xffffffff,%edi
        │        __u32 key = -1;
  40.54 │        mov    %edi,-0x4(%rbp)
        │        mov    %rbp,%rsi
        │
        │        add    $0xfffffffffffffffc,%rsi
        │ ➡      __u64 *v = bpf_map_lookup_elem(&test_map, &key);
        │        movabs $0xffff917fb02f4000,%rdi
  17.57 │        callq  *ffffffffe32c0324
        │        mov    %rax,%rdi
        │ ➡      if(v && *v == TEST_U64_VAL)
        │        test   %rdi,%rdi
  21.62 │        je     4a
        │        mov    $0x2,%eax
        │ ➡      if(v && *v == TEST_U64_VAL)
        │        mov    0x0(%rdi),%rdi
        │ ➡      if(v && *v == TEST_U64_VAL)
        │        cmp    $0x499602d2,%rdi
        │        je     4c
        │ 4a:    xor    %eax,%eax
        │      }
  14.87 │ 4c:    leaveq
        │        retq
```

- Same trivial function profiled on left and right, but with one difference
- **Look up element in array and compare to value.**
- Left side looks up valid index; right side looks up invalid index; annotations show the branch difference

```
Samples: 1K of event 'cycles', 4000 Hz, Event count (approx.): 333611469
bpf_prog_54ee63da2d576a5e_xdp_test  bpf_prog_54ee63da2d576a5e_xdp_test [P
Percent │      int xdp_test(struct xdp_md *ctx) {
        │        nop
        │        xchg   %ax,%ax
        │        push   %rbp
  4.00  │        mov    %rsp,%rbp
        │        sub    $0x8,%rsp
        │        mov    $0x7b,%edi
        │        __u32 key = TEST_U32_KEY;
 36.00  │        mov    %edi,-0x4(%rbp)
        │        mov    %rbp,%rsi
        │
        │        add    $0xfffffffffffffffc,%rsi
        │        __u64 *v = bpf_map_lookup_elem(&test_map, &key);
        │        movabs $0xffff917fb0ad4000,%rdi
  8.00  │        callq  *ffffffffe32ba060
        │        mov    %rax,%rdi
        │        if(v && *v == TEST_U64_VAL)
        │        test   %rdi,%rdi
        │        je     4a
        │        mov    $0x2,%eax
        │        if(v && *v == TEST_U64_VAL)
 24.00  │        mov    0x0(%rdi),%rdi
        │        if(v && *v == TEST_U64_VAL)
        │        cmp    $0x499602d2,%rdi
        │        je     4c
        │4a:     xor    %eax,%eax
        │      }
 28.00  │4c:     leaveq
        │        retq
```

```
Samples: 1K of event 'cycles', 4000 Hz, Event count (approx.): 323918153
bpf_prog_5f78d0a2aee6f57a_xdp_test  bpf_prog_5f78d0a2aee6f57a_xdp_test [P
Percent │      int xdp_test(struct xdp_md *ctx) {
        │        nop
        │        xchg   %ax,%ax
        │        push   %rbp
  5.41  │        mov    %rsp,%rbp
        │        sub    $0x8,%rsp
        │        mov    $0xffffffff,%edi
        │        __u32 key = -1;
 40.54  │        mov    %edi,-0x4(%rbp)
        │        mov    %rbp,%rsi
        │
        │        add    $0xfffffffffffffffc,%rsi
        │        __u64 *v = bpf_map_lookup_elem(&test_map, &key);
        │        movabs $0xffff917fb02f4000,%rdi
 17.57  │        callq  *ffffffffe32c0324
        │        mov    %rax,%rdi
        │        if(v && *v == TEST_U64_VAL)
        │        test   %rdi,%rdi
 21.62  │        je     4a
        │        mov    $0x2,%eax
        │        if(v && *v == TEST_U64_VAL)
        │        mov    0x0(%rdi),%rdi
        │        if(v && *v == TEST_U64_VAL)
        │        cmp    $0x499602d2,%rdi
        │        je     4c
        │4a:     xor    %eax,%eax
        │      }
 14.87  │4c:     leaveq
        │        retq
```

- Same trivial function profiled on left and right, but with one difference
- Look up element in array and compare to value.
- **Left side looks up valid index; right side looks up invalid index; annotations show the branch difference**

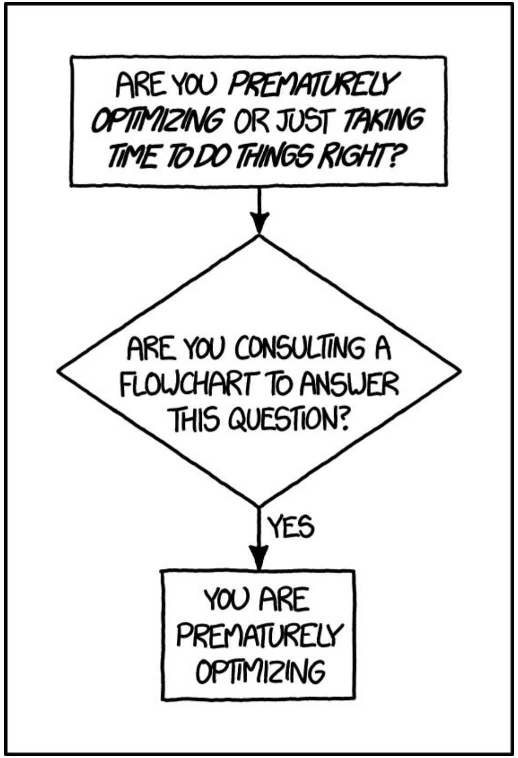
A few months of details...

1. Why is measuring XDP/BPF performance hard?
2. An approach for XDP performance analysis
3. **Outcomes**
4. Ongoing Work

verizon media

Outcomes

- **Does the methodology work?**
  - No flowcharts were consulted.
  - Running XDP accelerate load-balancer in production POPs
  - New areas of focus when evaluating changes to our systems
- **Sometimes you need to break problems into smaller chunks**
  - Flame graphs to understand how time is spent
  - Microbenchmarks to understand helpers and data structures
  - Code annotation to look for hotspots
- **Active Benchmarking is key for analysis success**
  - Make sure measurements make sense, tests do what they intend to do, and you can understand the results.
- **The introduction of BPF struct_ops and LSM hooks present opportunities for performance analysis beyond XDP programs**



xkcd 1691[5] (CC-BY-NC 2.5)

verizon media

A few months of details...

1. Why is measuring XDP/BPF performance hard?
2. An approach for XDP performance analysis
3. Outcomes
4. Ongoing Work

verizon media

Ongoing Work

- **BPF and networking stack are moving targets; new features every quarter to explore**
  - More building blocks to understand costs such as sleeping in BPF functions and freplace
  - kthread-based Rx network processing added in kernel 5.12
  - Investigate using BPF fentry/fexit hooks to monitor our XDP programs
  - `bpftool prog profile`
- **Measuring and understand jitter to ensure consistent performance**
  - Use kernel CPU isolation techniques (`CONFIG_NO_HZ_FULL` and `isolcpus=`)
  - BCC tools for observing hard and soft IRQ utilization with histograms
- **How to visualize instruction level sampling and annotation?**
  - Visualizing hot branch paths versus hot instructions
- **New consumer/producer based test harness added by Facebook last year to kernel selftests**
  - Build more robust benchmarks to simulate contention amongst

verizon media

Conclusion

- **We were reminded of the importance of Active Benchmarking**
  - The inaccuracy of CPU utilization metrics was unexpected but mitigated
  - A misconfigured evaluation system was spotted when evaluating microbenchmark data
- **We demonstrated how existing tools and new tools were used in our approach to XDP performance analysis**
  - Flame graphs, BPF_PROG_TEST_RUN, perf annotate, and BTF support in `perf`
  - Tackling a complicated or unfamiliar environment can be approached using existing methods and expanded on with new tools.
- **We hope this encourages others to dive deep into XDP and BPF analysis!**
  - This is a rapidly growing area of the Linux kernel
  - Good performance is a key factor to continued adoption

verizon media

Thank You!

- **Questions?**

- **Contact**
  - zjones@edgecast.com
  - https://zacharyjones.us
  - https://www.linkedin.com/in/zacharyhjones/

References

[1] XDP - IO Visor Project. https://www.iovisor.org/technology/xdp. Accessed: 2021- 04- 30.

[2] BPF and XDP Reference Guide — Cilium 1.10.90 documentation. https://docs.cilium.io/en/latest/bpf/. Accessed: 2021- 04- 30.

[3] Active Benchmarking. http://www.brendangregg.com/activebenchmarking.html. Accessed: 2021- 04- 30.

[4] LKML: Solio Sarabia: Re: Differences in cpu utilization reported by sar, emon. https://lkml.org/lkml/2018/6/20/1075. Accessed: 2021- 04- 30.

[5] Optimization. https://xkcd.com/1691. Accessed: 2021- 04- 30.