Assessing the Impact of Linux Networking on CPU Consumption

Politecnico di Torino

by **Davide Miola**

Contents

1. Introduction

1.1 Scope of this work and why it is important

WHAT

It is common knowledge that software networking is expensive*, but **just how much time do our servers' CPUs spend moving network packets around?**

WHY

- Discover possible optimizations
- Assess offloading potential
- …

HOW

No ready-to-use dedicated utility is available, so…
Let's build one!

*: but still increasingly relevant due to virtual networking *within* hosts

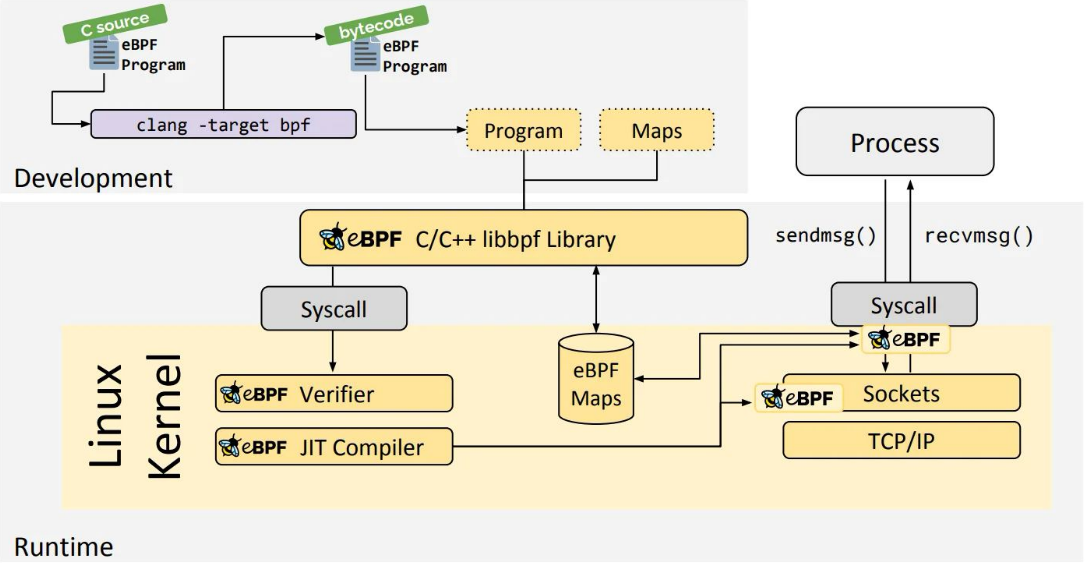
Netto

https://github.com/miolad/netto

1.2 Background

eBPF (extended Berkeley Packet Filter) is a technology of the Linux kernel that allows **dynamic injection and execution of user code into the kernel**

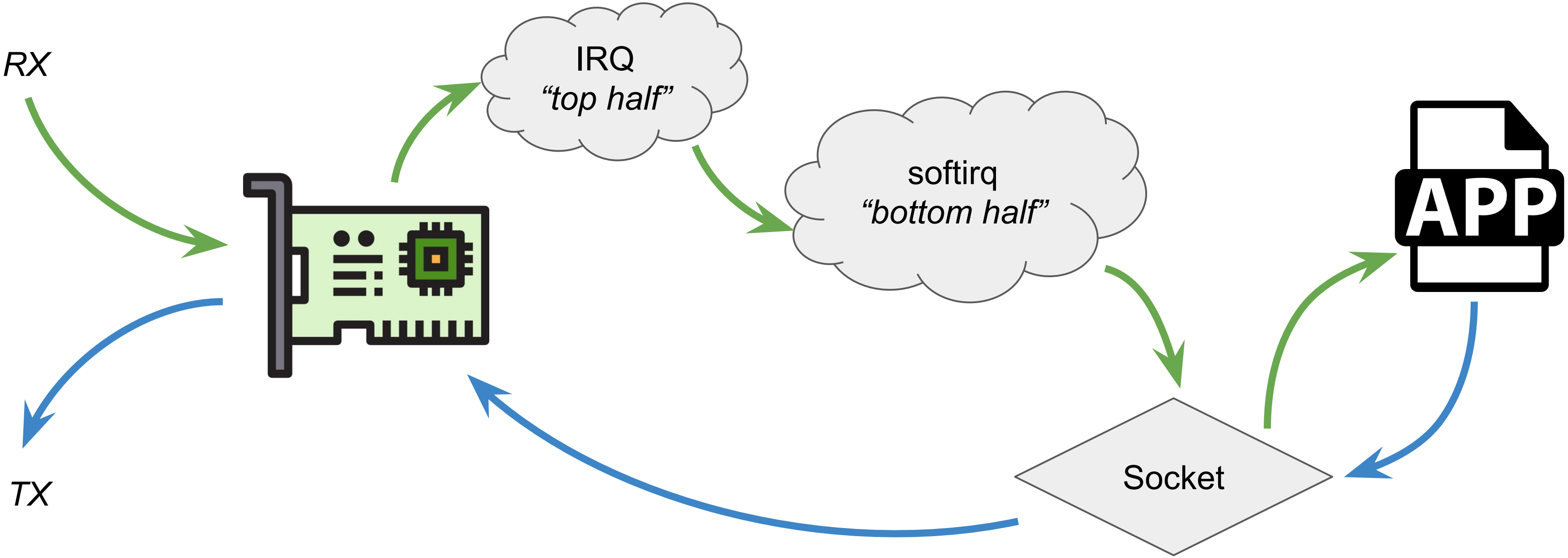
- **Fast**: Jitted code is run at near native speed
- **Safe**: Verifier ensures program correctness
- **Portable**: vCPU architecture is host agnostic (*mostly* [1])
- **Versatile**: Applications include tracing, networking data plane implementation, and more



---

1. https://lwn.net/Articles/779120/

1.2 Background

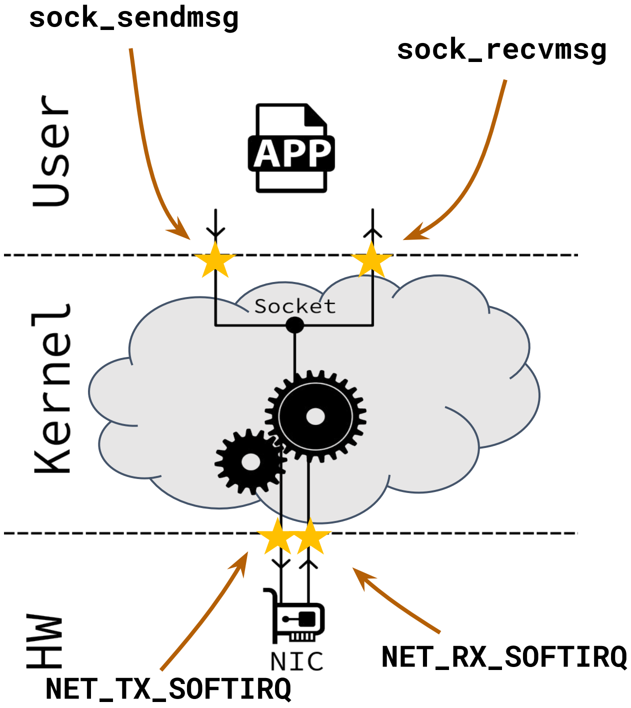


2. Design and Implementation

2.1 Basic algorithm overview

- Attach eBPF tracing probes to in-kernel networking entry points
- Measure on-CPU time as diff between entry and exit timestamps

Fast and efficient operation enables **real time, continuous monitoring**

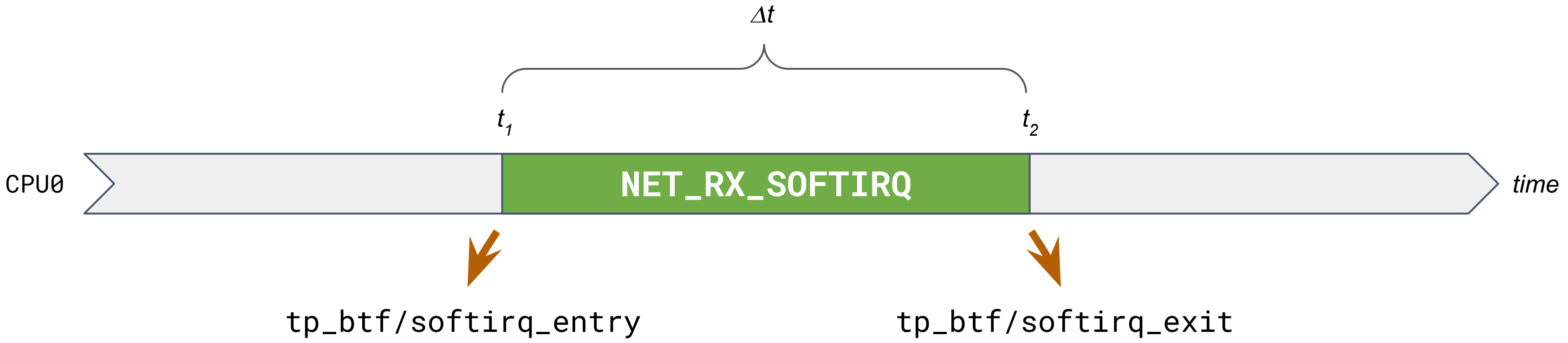


★ = network stack entry point

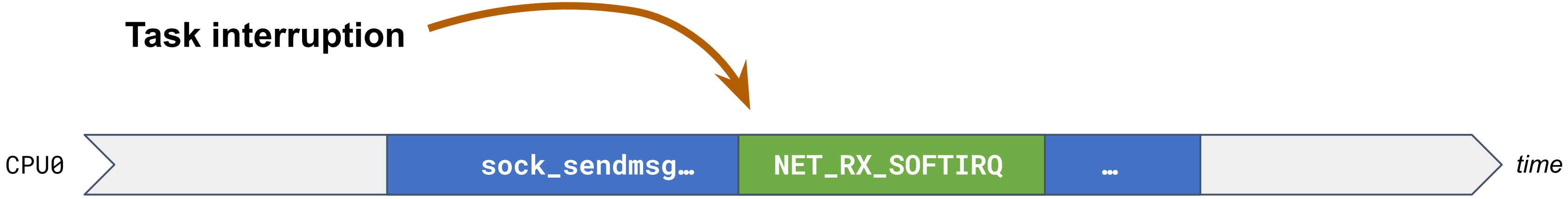
2.1 Basic algorithm overview

Event	Responsibility	eBPF *entry* ELF sec.	eBPF *exit* ELF sec.
`NET_RX_SOFTIRQ`	Network → Socket	`tp_btf/softirq_entry`	`tp_btf/softirq_exit`
`NET_TX_SOFTIRQ`	Flush TX queues	`tp_btf/softirq_entry`	`tp_btf/softirq_exit`
Socket recv ops	Socket → Application	`fentry/sock_recvmsg`	`fexit/sock_recvmsg`
Socket send ops	Application → Network	`fentry/sock_sendmsg`	`fexit/sock_sendmsg`

Typical operation:

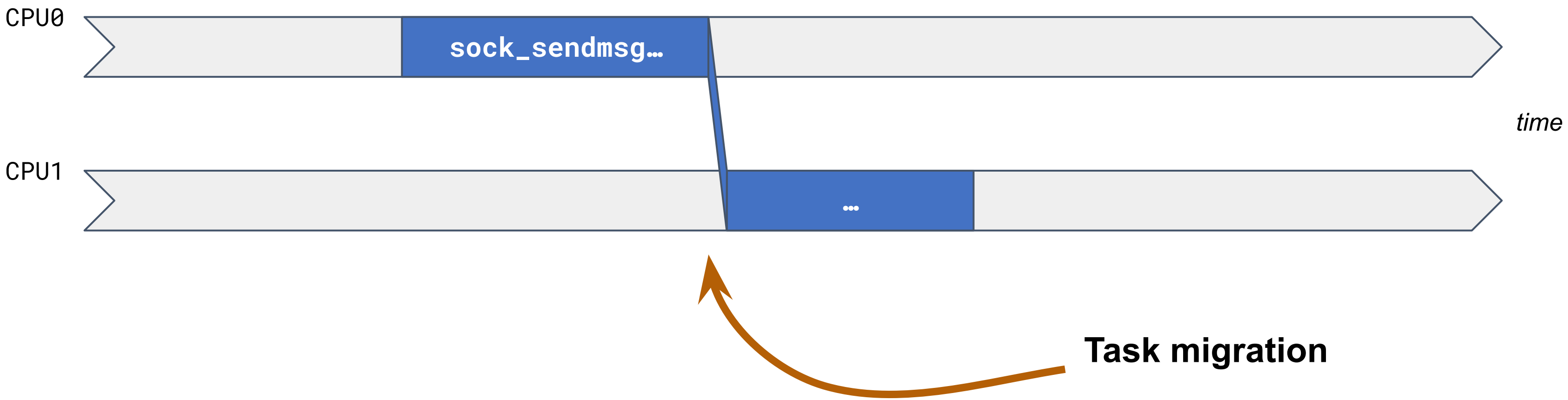


2.2 Handling switching of the execution context



1. Mark each kernel task with a flag identifying the currently running socket operation, if any (`BPF_MAP_TYPE_TASK_STORAGE` is perfect for this)
2. At every `tp_btf/softirq_entry` impersonate the socket operation's exit probe associated to the interrupted task's flag
3. Likewise for the `tp_btf/softirq_exit` tracepoint

2.2 Handling switching of the execution context



1. Instrument the `sched_switch` tracepoint
2. At every task switch, impersonate the outgoing task's exit probe and the incoming task's entry probe depending on their flag's value
3. Note that softirqs can not be preempted!

2.3 Event breakdown functionality

→ Bridging
→ Forwarding
→ Conntrack
→ NetFilter
→ Local Delivery
→ etc…

OBJECTIVE

Provide more in-depth insights of networking tasks

Breakdown of the `NET_RX_SOFTIRQ`'s contribution

Full Functions Tracking

Extend base approach to sub-functions of `net_rx_action`

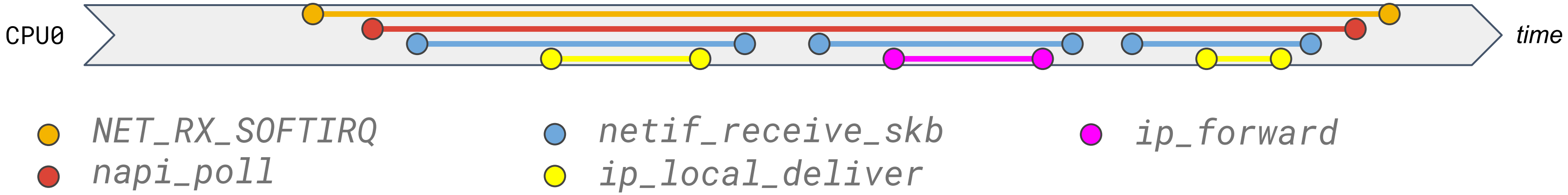
- Very complex eBPF code
- Very slow

Network Stack Sampling

Sample kernel-side stack trace regularly on all CPUs

- Tricky to move many traces to user-space
- Ultimately much faster and more elegant

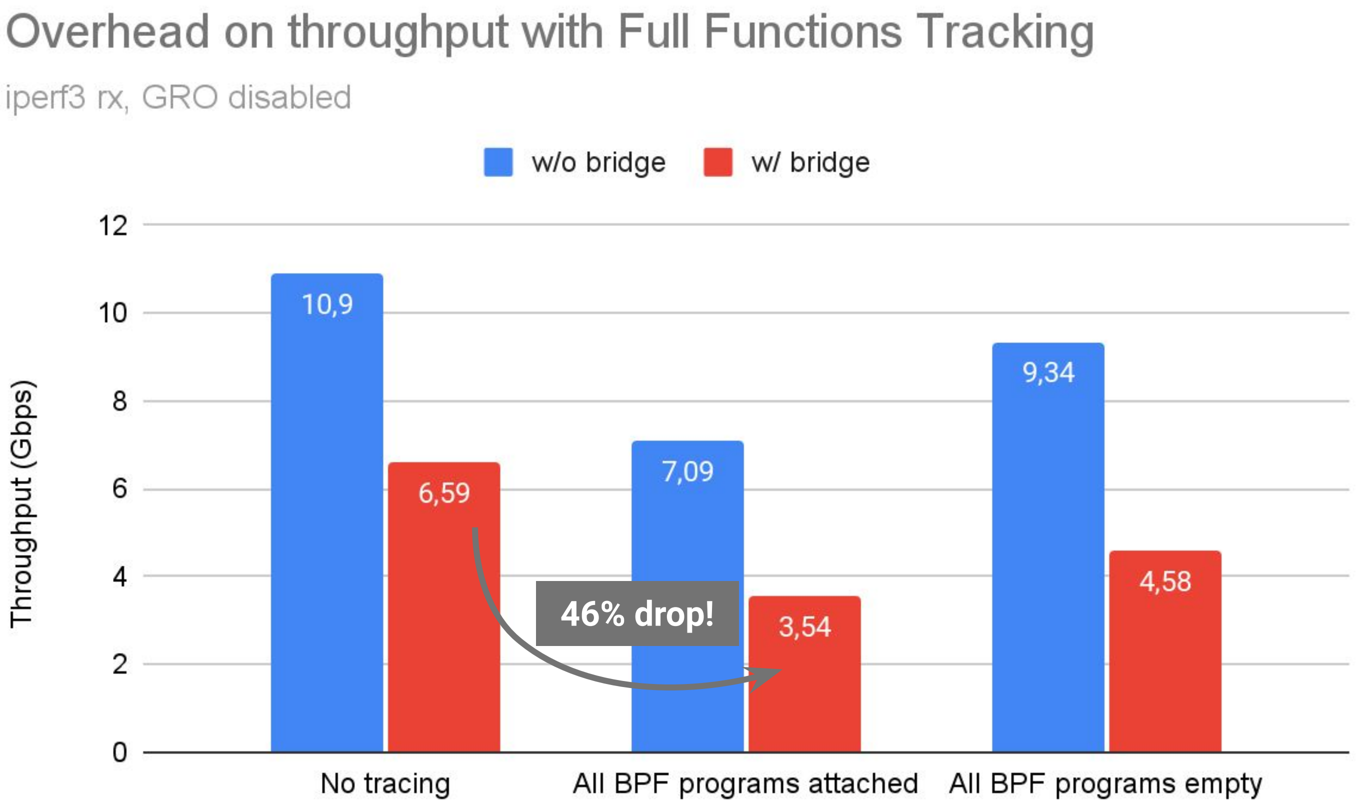
2.3.1 *Full Functions Tracking*



But…

complexity in the traced function hierarchy translates into **complex eBPF code**, and also **instrumenting per-packet functions is not a good idea** in high speed networks

2.3.1 *Full Functions Tracking*

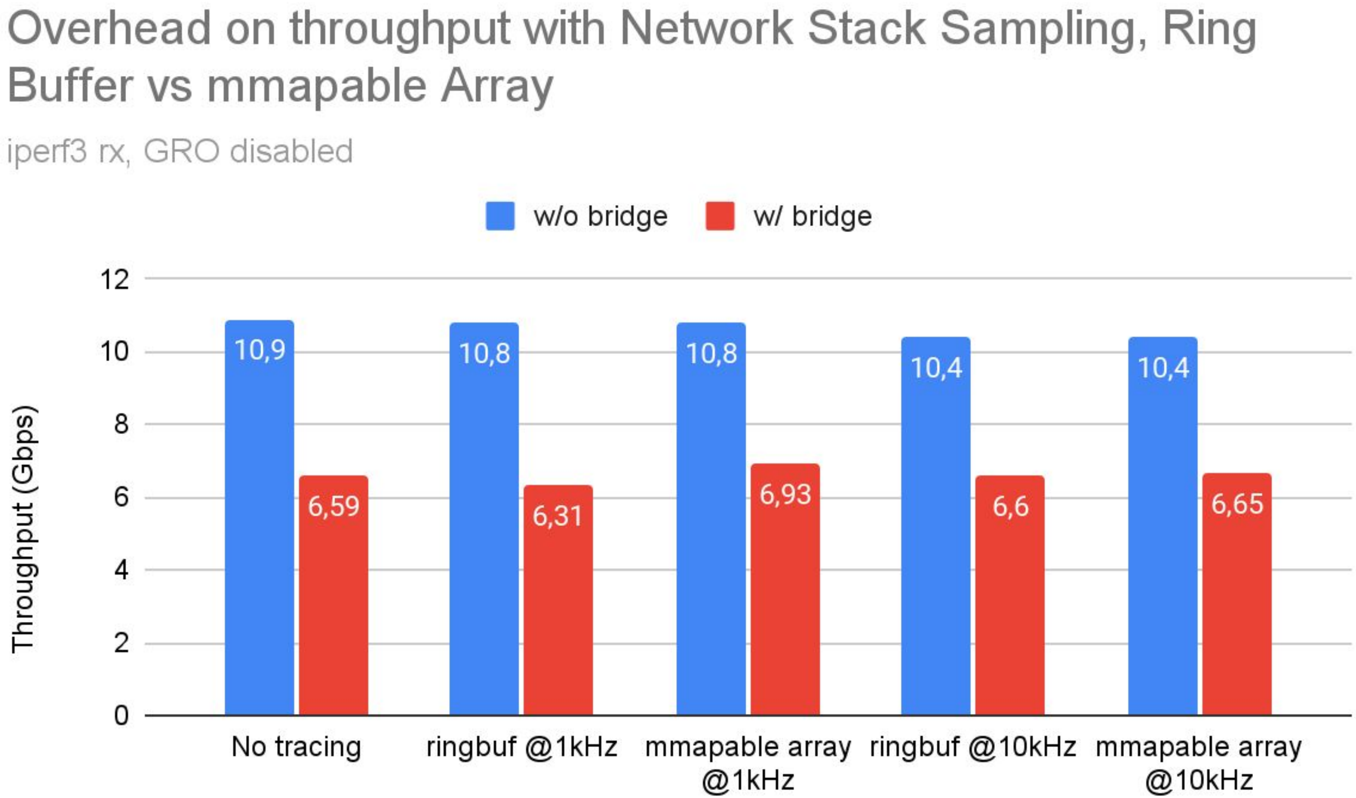


2.3.2 *Network Stack Sampling*

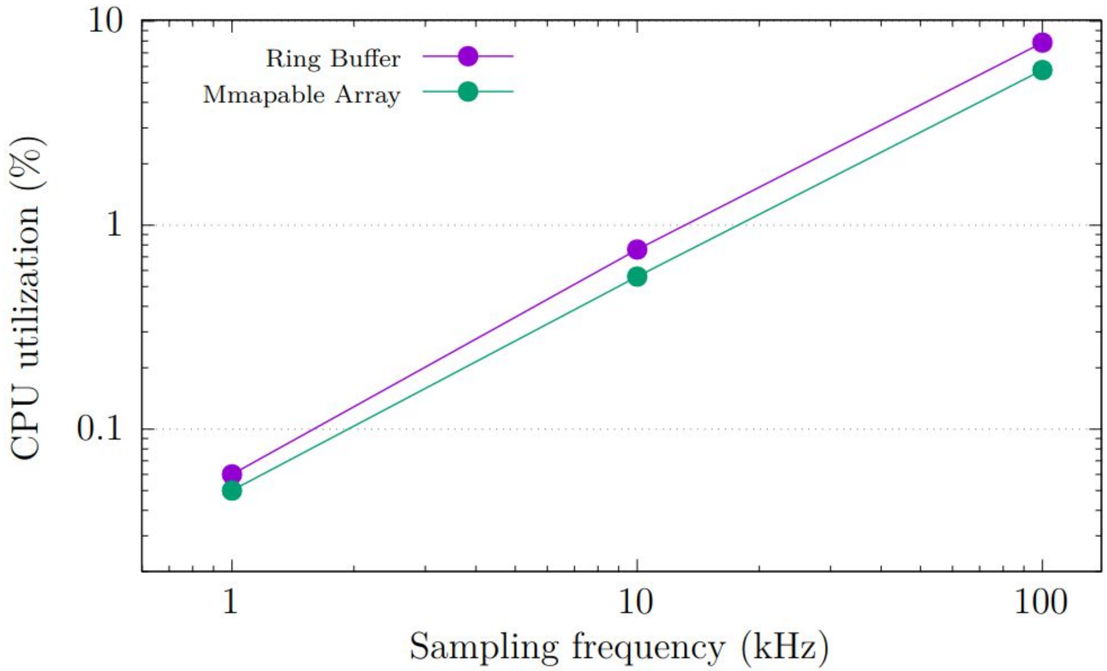
- `BPF_MAP_TYPE_STACK_TRACE` + hash map for counts
  - ✅ in-kernel trace summarization
  - ❌ requires two maps
  - ❌ no efficient method to retrieve them in user-space (multiple syscalls per stackid!)
- emulate stack trace map with hash map
  - ✅ in-kernel trace summarization
  - ✅ can copy whole map to user-space with one batch lookup
  - ❌ requires two stack dumps to get stackid in BPF
- BPF_MAP_TYPE_RINGBUF
  - ✅ “idiomatic” way to stream data from BPF to the user-space
  - ✅ only one stack dump per invocation
  - ❌ no in-kernel trace summarization
- mmapable `BPF_MAP_TYPE_ARRAY`
  - ✅ no syscalls to read traces in user-space
  - ✅ only one stack dump per invocation
  - ❌ no in-kernel trace summarization

w/ double buffering!

2.3.2 *Network Stack Sampling*



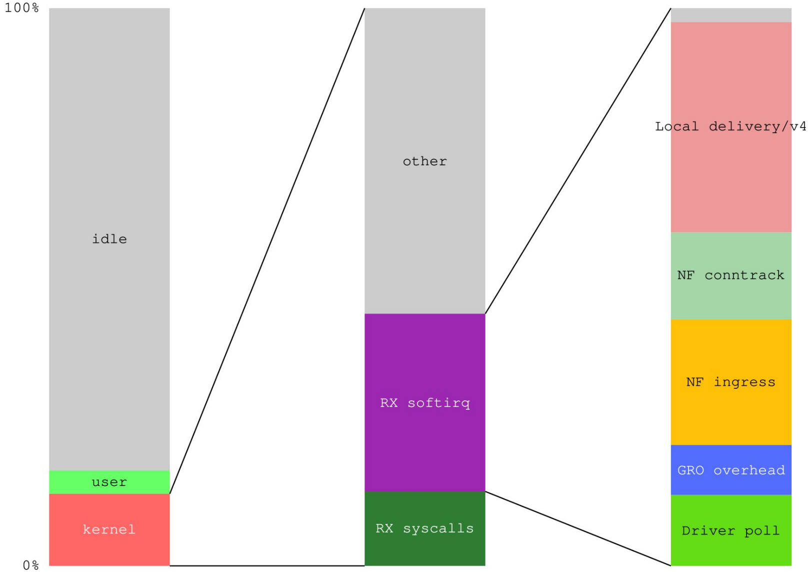
2.3.2 *Network Stack Sampling*



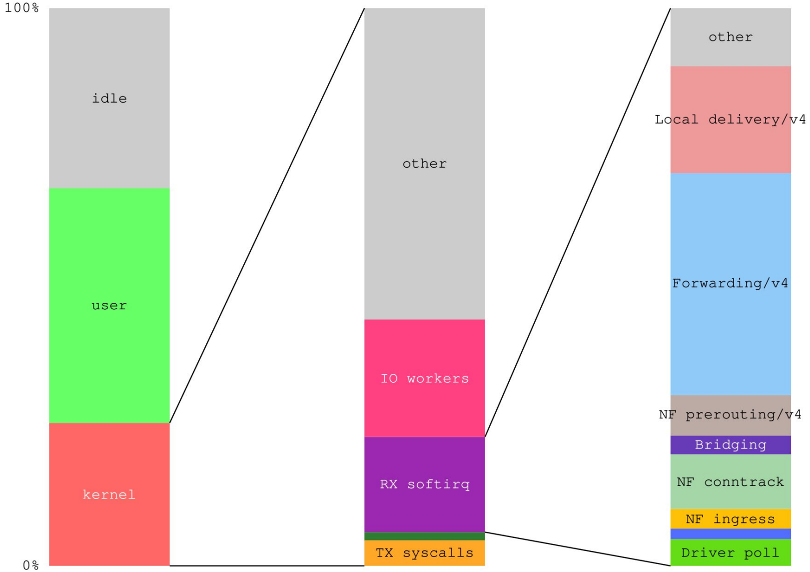
User-space CPU utilization for Network Stack Sampling with Ring Buffer and Mmapable Array backends.

3. Results and Validation

3 Results and validation



iperf3 UDP receive



Google's "Online Boutique" microservices demo

3 Results and validation

3 Results and validation

4. Conclusions

4 Conclusions

Current limitations

- Only measures in-kernel networking (i.e. no QUIC, TLS, or custom user-space data-planes)
- Ignores top-halves as well (wide range of implementations and minimal CPU consumption)

Future work

- Extend cost breakdown to more sub-events and, possibly, more top level entry points
- Explore an all-sampling measurement stack to further reduce overhead on high speed networks

Questions?