Assessing the impact of Linux networking on CPU consumption

Abstract

As network interfaces in the data-center get faster and faster, and an increasing portion of services is implemented in software, we wonder how many CPU cycles our servers are dedicating to handling network traffic. In fact, real world measurements always represent the first step to evaluate whether new optimizations are needed, in particular given the claim, coming from some SmartNIC vendors, that this cost can be up to 30% of the total amount of CPU cycles spent in a data center. This paper describes the design and functionality of a novel tool that enables in depth observation and monitoring of the Linux kernel's networking stack in real-time, through eBPF instrumentation of its main RX and TX entry-points. We also show how we can build a dynamic breakdown of the individual components on the fly while keeping the overhead down by collecting and analyzing CPU stack traces.

Introduction

Measuring a system's performance is key to understanding its bottlenecks and discovering possible optimizations. However, no dedicated tool is currently available to target the network sub-system of Linux-based hosts, which has traditionally forced system administrators to rely on broader, less focused diagnostic strategies to fine tune their servers.

Indeed, among the different network-oriented analysis techniques available in the literature, some require intrusive kernel modifications ([4], [5]) — and are thus hardly suitable for general-purpose measurements — while others rely on data-hungry hardware profilers, which make it impossible to deploy in a real-time, continuous manner (e.g., [3]).

This work presents *Netto* (`https://github.com/miolad/netto`), a new tool based on eBPF [8] which aims at filling this gap. This paper is structured as follows. Section Architecture and implementation introduces the design choices and implementation details of the tool; Section Results shows some preliminary results in controlled environments; finally, Section Conclusions and future work summarizes the obtained achievements and discusses possible future directions.

Architecture and implementation

In order to accurately measure the CPU time spent on networking tasks, we use eBPF probes placed at the entry and return addresses of the Linux functions that represent the main entry-points to in-kernel networking. By relying on eBPF for the instrumentation, we can ensure compatibility across distributions without requiring inconvenient patches to the kernel or device drivers, while maintaining an acceptable level of runtime overhead.

The specific networking entry-points that we identified — which we call "events" —, together with the associated C functions in the kernel source are the following:

- `NET_RX_SOFTIRQ` ([11, Version 6.4.8, Function `net_rx_action`, `net/core/dev.c`, Line 6661]): poll NAPI-based drivers and handle incoming raw packets; batches up to 300 *skbuff*s per invocation
- `NET_TX_SOFTIRQ` ([11, Version 6.4.8, Function `net_tx_action`, `net/core/dev.c`, Line 5018]): occasionally flush TX queues during high load scenarios
- socket receive ([11, Version 6.4.8, Function `sock_recvmsg`, `net/socket.c`, Line 1036]): common crossing point for syscalls such as `read`, `recv`, ...
- socket send ([11, Version 6.4.8, Function `sock_sendmsg`, `net/socket.c`, Line 742]): common crossing point for syscalls such as `write`, `send`, ...

We use the available `softirq_entry` and `softirq_exit` tracepoints for the first two events, in the sake of better efficiency, and rely on BPF trampolines [7] instead of kprobes for the remaining ones, for the same reason. The last two events also cover asynchronous variants of the input/output syscalls, including the new *io-uring* [12] interface.

Every *exit* BPF program accumulates the difference in its invocation timestamp with that of its respective *entry* into an event-specific, per-cpu, monotonically increasing nanosecond counter that is shared with the user space controller. The control loop runs at a frequency of 2 Hz, reads the most up-to-date values from the BPF map and updates the user-facing report.

To correctly handle preemption, we mark each task with a flag that encodes the currently running event, and instrument the `sched_switch` tracepoint to be notified of task switches: each such occurrence is thus treated as an *exit* from the active event on the outgoing task and a concurrent *en-*

try into the active event on the incoming task. Note that this works because Linux' softirqs are not preemptible.

Event breakdown

One of the key capabilities of *Netto* is providing an estimated breakdown of the cost of the network "events" into their main components. With this feature — currently implemented for the `NET_RX_SOFTIRQ` event — we can approximate the cost of the individual network functions of a Linux host, such as bridging or forwarding.

We achieve this by performing real time profiling of the kernel threads: we instrument a Linux *perf-event* [6] with an eBPF program to capture the kernel-side CPU stack trace on all cores at a set frequency (1 kHz has been shown to provide both very low overhead and a satisfactory perceived accuracy); these traces are then efficiently brought to the user-space controller through a double-buffered *mmap*ed array map, where they are matched against a set of predefined kernel symbols.

By introducing an approximation in the form of a sampling frequency, we are able to measure per-packet metrics, which would be prohibitively expensive for high load scenarios if done with traditional eBPF probes instead.

Results

To demonstrate the quality of the analysis, we tested the tool against a set of controlled workloads. Our testbed consists of two identical *Intel Core i7 6700*-based machines running kernel 5.15 LTS, directly connected at 40 Gbps through a pair of *Intel XL710* NICs.

iperf3 receive

The two systems were set up to perform *iperf3* [1] TCP and UDP throughput tests, and *Netto* was loaded on the receiver.

Figure 1 shows a snapshot of the real time generated CPU cost diagram during the transfers. In the case of TCP, virtually all kernel time (which is represented by the central bar) is spread between `reads` and the `NET_RX_SOFTIRQ`. Conversely, for UDP, a large portion of the kernel's total time is reported as "*other*", which in this case hides complementary system calls found in most I/O loops like `select`, and the generic syscall entry and exit overhead.

The reason why these components have a much larger impact in UDP is syscall frequency: the UDP transfer pushes about 700k system calls every second (around seven times as many as those generated by the TCP test), which also constitutes a severe bottleneck to the achieved throughput.

The bandwidth overhead associated to the real time analysis, as measured at the receiver, is about 6% for UDP. In TCP, where the only bottleneck is the physical link's speed, no negative impact could be observed.

Google's "*Online Boutique*" microservices demo

For a test more representative of the typical web server workload, we deployed Google's "*Online Boutique*" [2] microservices demo on a *KinD* [9] cluster and exposed the web frontend through a *MetalLB* [10] load balancer. On the other machine, Google's provided `loadgenerator` microservice

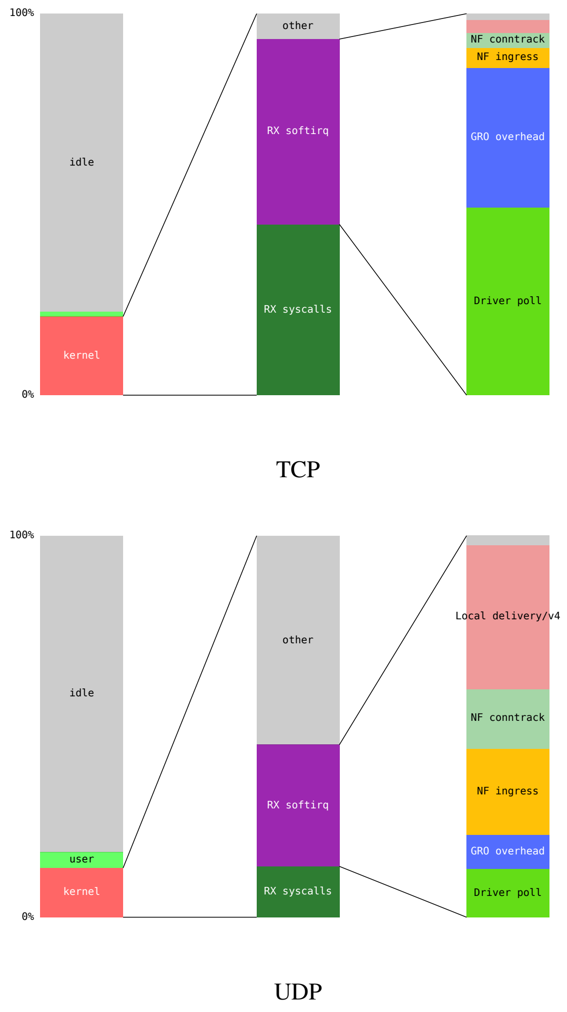

Figure 1: Measured networking cost during an *iperf3* TCP (top) and UDP (bottom) receive tests.

was used to stress the cluster at a rate of about 1000 requests per second. Figure 2 depicts the results.

The graph indicates that the overall CPU time is distributed approximately evenly among user and kernel modes, and that in-kernel networking only accounts to a relatively limited portion of the total system time; otherwise, the kernel spends significant time in a *io-uring* `SQPOLL` burner thread (indicated as "*IO workers*" in the graph).

Conclusions and future work

In this paper we presented a new, eBPF-based tool to aid monitoring and diagnosis of the networking subsystem in Linux hosts; we have also shown how a sampled estimate approach can allow the effective observation of hot paths in the kernel code without a detrimental impact on system performance.

On the other hand, the design choice of only considering kernel networking means that any user-space network function is excluded from the analysis. This includes QUIC, TLS

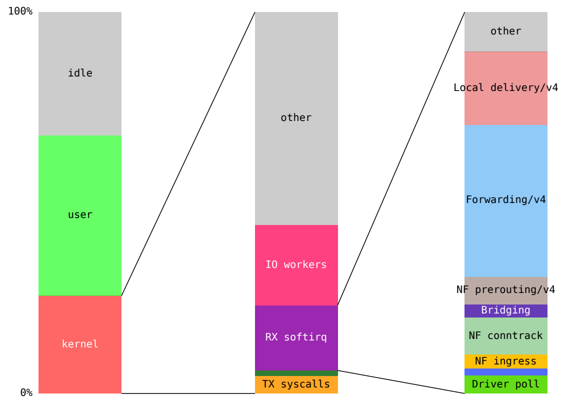

Figure 2: Measured networking cost while stress-testing Google's "*Online Boutique*".

and DPDK, as well as application-level proxies that are becoming so common in the cloud-native world.

Finally — considering possible future developments — one of the most promising perspectives involves hooking the dynamic analysis provided by our tool to system management applications that can utilize the current networking load to drive resource allocation algorithms.

References

[1] Dugan, J.; Elliott, S.; Mah, B. A.; Poskanzer, J.; and Prabhu, K. iperf3 throughput testing tool. `https://iperf.fr/`. Accessed: 2023-08-15.

[2] Google. "*Online Boutique*" microservices demo. `https://github.com/GoogleCloudPlatform/microservices-demo`. Accessed: 2023-08-16.

[3] Haecki, R.; Mysore, R. N.; Suresh, L.; Zellweger, G.; Gan, B.; Merrifield, T.; Banerjee, S.; and Roscoe, T. 2022. How to diagnose nanosecond network latencies in rich end-host stacks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 861–877. Renton, WA: USENIX Association.

[4] Peter, S.; Li, J.; Zhang, I.; Ports, D. R. K.; Woos, D.; Krishnamurthy, A.; Anderson, T.; and Roscoe, T. 2014. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 1–16. Broomfield, CO: USENIX Association.

[5] Rizzo, L. 2012. netmap: A novel framework for fast packet I/O. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 101–112. Boston, MA: USENIX Association.

[6] Starovoitov, A. BPF perf-event Linux patch. `https://lore.kernel.org/lkml/1472680243-1326608-3-git-send-email-ast@fb.com/`. Accessed: 2023-08-14.

[7] Starovoitov, A. BPF trampolines Linux patch. `https://lore.kernel.org/bpf/20191114185720.1641606-5-ast@kernel.org/`. Accessed: 2023-08-14.

[8] The eBPF community. eBPF. `https://ebpf.io/`. Accessed: 2023-07-17.

[9] The Kubernetes Authors. KinD: Kubernetes in Docker. `https://kind.sigs.k8s.io/`. Accessed: 2023-08-16.

[10] The MetalLB Contributors. MetalLB. `https://metallb.universe.tf/`. Accessed: 2023-08-16.

[11] Torvalds, L. Linux. `https://kernel.org`. Accessed: 2023-08-09.

[12] Wikipedia contributors. io_uring - *Wikipedia, The Free Encyclopedia*. `https://en.wikipedia.org/wiki/Io_uring`. Accessed: 2023-22-08.