Measuring the cost of the Linux network stack in real-time

Measuring the Cost of the Linux Network Stack in Real-Time

Davide Miola, Fulvio Risso, Federico Parola
Department of Computer and Control Engineering
Politecnico di Torino
Turin, Italy
Emails: {davide.miola, fulvio.risso, federico.parola}@polito.it

***Abstract*—As network interfaces in the data center get faster and faster, and an increasingly big portion of the services is implemented in software, we must wonder just how much time our servers' CPUs are spending handling network traffic.**

This paper explores the feasibility of measuring the cost of the entire in-kernel network stack in real-time on production systems by relying on the eBPF tracing capabilities instead of utilizing custom logic or kernel patching. We describe two methods that have been attempted, respectively based on an "exact" instrumentation of the stack and sampling, along with the advantages and defects of each approach.

***Index Terms*—Linux, networking, eBPF, tracing, observability**

I. Introduction

Modern computer systems heavily rely on efficient networking capabilities to handle the increasing demands of data transfer and communication. Within this context, the kernel's networking stack plays a vital role in both (*i*) facilitating network communication, and (*ii*) ensuring data integrity. Therefore, as network traffic continues to surge with the widespread adoption of IoT (Internet of Things) devices [1] in the attempt to create the so-called Smart Cities and Societies, it is of paramount importance to understand and optimize the kernel's overhead imposed by the networking stack to achieve optimal system performance to deal with the massive amount of generated data.

To address this challenge, emerging technologies such as eBPF [8] (extended Berkeley Packet Filter) have gained significant attention due to their ability to perform dynamic and non-intrusive analysis of kernel behavior. By providing a safe and efficient means to extend and customize the kernel's functionality, eBPF has opened up new possibilities for deep analysis of the networking stack and its associated performance overhead.

The primary objective of this paper is to explore the use of eBPF as a powerful tool for tracking and quantifying the overhead induced by the networking stack of the Linux kernel. By utilizing eBPF, we can collect fine-grained data and gain valuable insights into the behavior of the kernel during network operations. This approach allows not only to pinpoint performance bottlenecks but also to identify and possibly optimize code and infrastructure inefficiencies, ultimately leading to improved system performance and resource utilization.

In this paper, we present a comprehensive comparison of techniques for utilizing eBPF to track kernel overhead in the networking stack, following their implementation and testing in *Netto* (https://github.com/miolad/netto), a new tool to aid system administrators to diagnose the CPU consumption of the networking subsystem of Linux-based operating systems. These techniques attempt to isolate the networking cost of the kernel by respectively instrumenting the specific kernel functions responsible for implementing such networking functionality (thus obtaining an "exact" evaluation), and sampling the CPU kernel-side call stack to profile the networking code. We will also discuss the collection and analysis of performance data, and the interpretation of the results. Through experiments, we extensively evaluate (*i*) the performance overhead of the proposed solutions, and (*ii*) the measurement accuracy with respect to each other.

The remainder of the paper is organized as follows. Section II summarizes the related work. Section III introduces the key concepts of the Linux networking stack. Section IV describes the architecture and implementation details of instrumenting the Linux network stack using eBPF. Section V discusses processing overhead and measurement accuracy of the proposed methodologies, and also evaluates the tool against a set of test cases which showcase the typical results achievable through *Netto*. Section VI analyzes some of the current limitations of out design and considers future development opportunities. Finally, Section VII concludes the paper.

II. Related Work

The need to measure costs and overheads in a kernel's network stack is quite common for research work on techniques to mitigate bottlenecks and improve performance, but no unified measurement framework exists that can satisfy all the usual requirements of accuracy and transparency. Nevertheless, the literature proposes several techniques to profile specific sections of the networking stack, but in no case the suggested method proves exhaustive in measuring the entire networking layer of the Linux kernel. In the following we will present a few of them to discuss advantages and limitations.

As part of the *netmap* project, Rizzo [6] adopted a custom solution to obtain the baseline performance profile of FreeBSD's `sendto` system call: a userspace program was used to average out the execution time of the designated syscall, and by repeatedly patching the kernel to short-circuit the syscall at different stages of its execution path, a detailed breakdown of its cost could be obtained. Such a solution is appropriate only for static analysis of a system's performance, but it is clearly not suitable for real-time or continuous monitoring, as it requires multiple intrusive kernel modifications to collect a single measurement. Moreover, this approach does not take into account scheduling, and is therefore unreliable on loaded systems. Apart from these drawbacks, Rizzo's solution would still not contend *Netto* due to the limited extent of the measurement and inadequate scalability.

Two years later, Peter et al. [5] used a similar technique to profile the UDP path of the Linux network stack. Unlike [6], however, measurements were taken by timestamping meaningful events directly in the kernel, rather than averaging out short-circuited versions of the syscall of interest. This approach still requires patching the kernel but avoids breaking its functionality while collecting the samples. Otherwise, the scope remains that of a focused, one-time measurement: once again, extending this concept to the whole network stack would need a massive kernel patch which would be incompatible with the Plug-and-Play nature that *Netto* strives for. Also, timestamping per-packet hot paths would likely prove challenging due to the associated overhead; *Netto* overcomes these complications by sampling these otherwise prohibitive functions for a fixed, configurable cost.

NSight [3] is a recent tool that allows to diagnose latency deviations of network packets in end-hosts which claims both very high precision and negligible overhead. It manages this by utilizing the hardware profiler Intel-PT [4] to follow a network packet's journey through the network stack from the NIC to the application (or vice-versa), and build its "lifetime". This is later analyzed in relation to that of neighbouring packets to reveal anomalies and their causes. Because of the Intel-PT dependency, NSight collects 600 MB to 1 GB of compressed profiling data every second, which hinders its chances of deployment as a real-time monitoring solution. Furthermore, NSight requires a — albeit small — patch to the mainline Linux kernel to obtain NIC timestamps at kernel to userspace hand-off boundaries, making it cumbersome to adopt on production systems.

In this paper, we propose two methods based on eBPF which do not require patching the kernel, while remaining lightweight enough to support continuous operation, with the ambition of negligible system overhead.

III. Linux Networking Summary

The Linux network stack is the portion of the kernel in charge of handling (and moving) network data between (*i*) one network interface card (NIC) and the target application (and vice-versa), and (*ii*) a NIC to another (for bridging and forwarding), regardless of whether network interfaces are physical NICs or virtual devices that live entirely in software. While the first path looks like the most common in servers, it is actually the second one that is even more important, given that modern data centers massively rely on bridging and routing functions, e.g. to move network data among different virtual machines or containers within the same host.

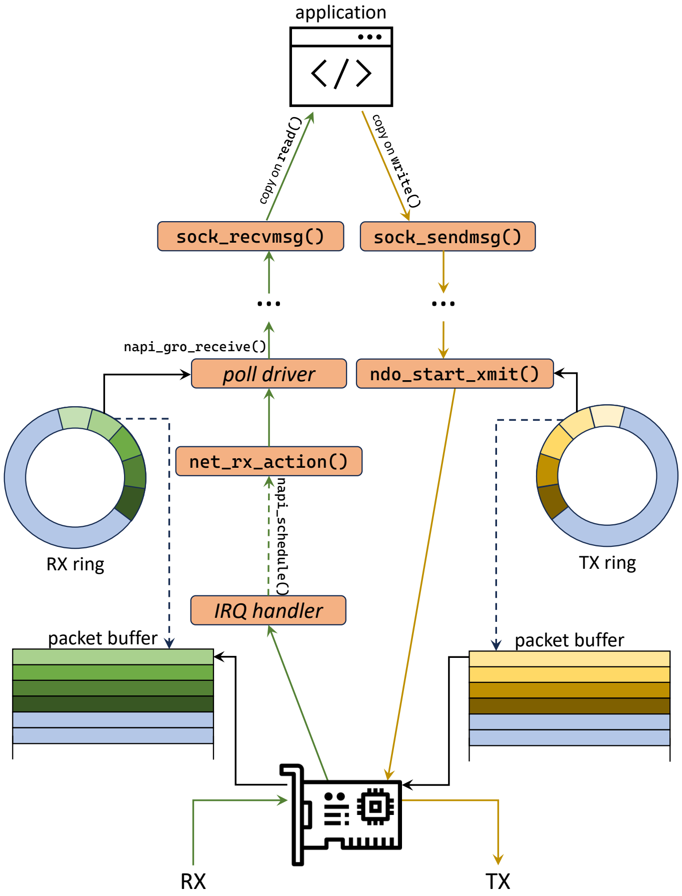

Fig. 1: High level overview of the Linux kernel's network stack software architecture. The figure depicts both the receive (*left*) and transmit (*right*) paths.

The following subsections detail the journey of a typical network packet as it is received by an interface until it gets delivered to the destination application.

A. Network to socket

When a packet is received on a physical interface, it is DMAed to a driver-owned buffer in kernel memory, and an interrupt is raised by the hardware. Due to its complexity, the packet handling stack can not all be contained in the Interrupt Service Routine (ISR) of the NIC driver, and it is thus split into a *top half* and a *bottom half*. This architecture, which is common across a variety of device drivers, allows to preserve system responsiveness by keeping the high-priority ISR (top half) compact and fast, while delegating the bulk of the computation to a deferrable function (bottom half), usually run in a lower priority context.

To this end, Linux provides *deferrable functions* as a means for drivers to schedule code to be executed at a later time, thus allowing to move expensive computations out of the constrained interrupt context; this makes them ideal for implementing typical bottom halves. In modern versions of the Linux kernel, several *deferrable function* implementations exist; among them, *softirqs* are an ideal fit for the networking layer of the operating system, thanks to their favourable concurrent execution behavior and low overhead due to the tight integration into the kernel's design. By contrast, *tasklets* (an abstraction on top of softirqs) trade performance with simplicity by disallowing parallel execution over multiple CPU cores, and *work queues* are controlled by the system's global task scheduler, meaning that an hypothetical network stack implementation based on one would result in unpredictable and inconsistent packet latency. Indeed, two of the ten currently allocated softirq types are related to networking: `NET_RX_SOFTIRQ` and `NET_TX_SOFTIRQ`.

In virtually all modern NIC drivers, which are compliant with the NAPI[1] interface, the top half wraps a call to `napi_schedule()`, which raises the `NET_RX_SOFTIRQ` on the local CPU and registers the NIC's driver for polling (Figure 1, left side). For each invocation, the `NET_RX_SOFTIRQ` will poll all registered drivers (within its budget constraint of 300 packets or 2 *jiffies* by default), by calling their provided `napi_poll()` virtual function. At this point, the typical NIC driver will perform a "cleanup" of the RX descriptor rings, a process which can be broken down into the following high level overview, for each extracted packet:

1) Run any `XDP_NATIVE` eBPF programs, if supported.
2) Wrap the packet into an *skb*, populating fields such as *protocol* and *VLAN*.
3) Submit the packet to the upper networking layer with `netif_receive_skb()`, `napi_gro_receive()` or similar (the specific function used depends on the driver's capabilities, *not* on the NIC or its current configuration).

Once the common networking layer of the Linux kernel is reached, the skb is managed based on its L3 protocol by delivering it to the appropriate *protocol handler* (like `ip_rcv()` for IPv4 traffic). Similarly, bridging is implemented by a specific *receive handler* — `br_handle_frame()` — which is run on every frame received on an interface that is part of a software bridge. Crucially, a bridge can "pass a frame up" when an skb targets the bridge itself with a nested call to `netif_receive_skb()`.

B. Socket to application

To provide networking services to userspace processes, Linux adopts the traditional socket API. Sockets represent the interface between applications and in-kernel network functions, and through them users can send and receive data.

The scope of the `NET_RX_SOFTIRQ`, for traffic intended for the local host, ends after dispatching the packet's payload to the appropriate socket's receive buffer. Note however that not all incoming packets must be delivered locally. Some, for example, might be transmitted on output interfaces due to bridging or forwarding, or because of ACK generation, while others may be dropped because of filtering or other reasons.

The traditional way of interacting with socket objects in Linux has been the system call interface. Syscalls such as `read` and `recv` may be used to retrieve data from the socket's kernel-side buffer and copy it to the user-provided address, completing the packet's journey. If no data is currently available, most variants of the aforementioned system calls will put the user process in "io-wait", blocking it until there is something to read. Some asynchronous versions of these functions have been proposed throughout the years, like the promising new *`io_uring`* [11] interface.

C. Data transmission

The network subsystem of Linux is, by nature, fairly asymmetric. This is because the reception of an incoming network packet is inevitably an asynchronous event with respect to the local computer system. Consequently, its management is delegated to a mechanism of (hard and soft) interrupts, until the received packet is eventually assigned to a process via the owning socket.

In the opposite direction, the picture is generally much simplified, as socket writes represent the primary entry point into the network stack for locally generated outbound traffic. For a usual blocking call like `write` or `send`, the entire TX network stack is executed within the syscall, which brings data from the user-space buffer to the output NIC driver (Figure 1, right side). It must be noted, however, that this simplistic dissertation does not take into account complications specific to individual protocols, like in the case of TCP, which uses deferrable tasklets and timers to aid its congestion and flow control implementations.

Lastly, a `NET_TX_SOFTIRQ` is activated for the deferred transmission of pending packets in case the target interface is still busy sending previous data. In any case, this last softirq hardly matters in the overall picture for CPU consumption due to the infrequent nature of its invocations.

IV. Architecture and Implementation

In this section, we delve into the two tracing models implemented in *Netto*, explaining their software architecture and design choices, first by tackling the solution based on an exact analysis, then outlining the sampling approach.

A. Full Functions Tracking

An intuitive and general method to measure the CPU time of any software component of the Linux kernel is to surround it with eBPF probes attached to the entry and return addresses of the functions responsible for implementing it. The event's run time can then be derived as the difference in the invocation timestamps of the eBPF programs on a per-CPU basis. By relying on eBPF for the instrumentation, we can ensure compatibility across distributions without requiring inconvenient patches to the kernel or device drivers, while targeting an acceptable level of runtime overhead.

[1] NAPI (formerly *New API*) is the event handling mechanism used in the Linux networking stack. https://docs.kernel.org/networking/napi.html

In the case of the Linux network stack, the same concept can be applied by first identifying the main entry points into the networking code. Following the outline of the network stack's architecture described in the previous Section III, we determined the following functions to be the key locations where eBPF tracing was to be installed:

- **`NET_RX_SOFTIRQ`**: Polls NAPI-based physical and virtual network drivers and handles incoming raw packets until dispatchment to local sockets; it batches up to 300 *skbuffs* per invocation by default.
  This softirq is responsible for most of the receive-side network stack, as well as *reactive* outbound traffic (i.e. transmitted packets that are produced in reaction to some received message, such as TCP acknowledgments or ICMP echo replies) and forwarded outbound traffic (like for L2-bridging and L3-forwarding network functions).
- **`NET_TX_SOFTIRQ`**: Occasionally flushes transmission queues for busy networking drivers during high load scenarios.
  As opposed to the RX softirq, the `NET_TX_SOFTIRQ` generally has a limited CPU footprint.
- **Socket receive operations**: User-callable functions to terminate a receive operation. These include system calls like `read` and `recv`, which ultimately copy available received data from in-kernel socket buffers to user-provided memory for consumption.
  The identified C function associated with this event is `sock_recvmsg()`, which acts as a common crossing point for the multiple available socket read paths.
- **Socket send operations**: User-callable functions to trigger transmission of data through a socket object. Unlike read operations, a write will usually account for most of the transmission side of the Linux kernel's networking stack. These include system calls such as `write` and `send`. The identified C function associated with this event is `sock_sendmsg()`, which acts as a common crossing point for the multiple available socket write paths.

Table I summarizes the chosen eBPF program types and attach points for the four abovementioned events in the format of their binary ELF sections[2]. The decision is attributable to an effort at minimizing in-kernel overhead by the eBPF instrumentation; for this reason, static tracepoints were preferred whenever available, and trampoline-based `fentry` and `fexit` programs [7] were used otherwise. Note that, as both softirq events map to the same set of tracepoints, the two are distinguished by filtering over their "`vec`" arguments, which identifies their specific type. It is also worth mentioning that the last two events are hit by any of the similar system calls, including their asynchronous analogues and the networking *opcodes* of the `io_uring` interface.

The described set of eBPF programs allows *Netto* to measure the on-CPU time for each of the four network events independently. Every *exit* program calculates the difference in its invocation timestamp with that of its respective *entry* and accumulates it into an event-specific counter. All four counters live in a `BPF_MAP_TYPE_PERCPU_ARRAY` map, enabling them to (*i*) retain per-CPU resolution over the measured usage metrics, and (*ii*) be exported to the user space controller for consumption. The controller runs with a default period of $500\,\mathrm{ms}$, reads the most up-to-date values from the map — which it compares with the previous iteration's counters —, and then updates the user-facing report. In particular, the total amount of CPU time that Linux spent inside its networking stack is computed as the sum of the on-CPU time of the individual events. The default control loop invocation frequency of 2 Hz was chosen as a good compromise between a satisfying temporal resolution of the output data and an acceptable level of CPU overhead, although the initial concerns about elevated system load due to the user-space controller have not materialized.

We are now going to discuss some of the most prominent complications that arise from this design, which require ad-hoc and careful handling to avoid data races and measurement inconsistencies (§IV-A1). Next, an additional feature is explored in order to attempt a breakdown of the obtained cost figure, enhancing *Netto* with the ability to estimate the CPU consumption associated with the individual network functions like bridging and forwarding (§IV-A2).

1) Handling switching of execution context: The design presented so far works well in the common case, but it is subject to errors in preemptible kernels in two separate scenarios: (*a*) network softirqs *can* interrupt tasks on which other monitored events are running, and (*b*) the task scheduler could preempt or migrate monitored tasks.

a) Softirqs interrupting other tasks: Linux softirqs can be run in IRQ context, and can thus interrupt any user process, including those where an identified socket operation (i.e., send and receive system calls) is running. This occurrence is especially common during high network load situations, and leads to the over-estimation of the CPU utilization of the interrupted task. It is worth pointing out that the opposite case, where a `NET_RX_SOFTIRQ` or `NET_TX_SOFTIRQ` is interrupted by a socket operation, is not possible since softirqs are hosted on non-preemptible execution contexts.

The adopted solution consists in associating to each encountered kernel task a pair of bits allocated to store information about what socket operation is currently executing, if any. These bits are set at every entry and exit to the `sock_recvmsg` and `sock_sendmsg` events. Softirq entries will then behave like an exit from the socket recv/send event whose id is stored in the task data, and vice-versa, softirq exits will act as the corresponding socket event's entry, updating the global entry timestamp.

b) Migratable tasks: The second issue is related to tasks being preempted out of the CPU by the Linux scheduler, possibly migrating them to different cores. Since metrics exported by the eBPF layer have a per-CPU resolution, any such migration

[2] An eBPF program's section in its compiled ELF binary determines its type and concrete attach point. See https://docs.kernel.org/bpf/libbpf/program_types.html for reference.

TABLE I: eBPF programs' ELF sections for the four identified networking entry-points in the exact measurement approach.

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 operations	Socket → Application	`fentry/sock_recvmsg`	`fexit/sock_recvmsg`
Socket send operations	Application → Network	`fentry/sock_sendmsg`	`fexit/sock_sendmsg`

negatively influences the measurement's accuracy, as the exact on-CPU time of a migrated task can not be correctly estimated by its eBPF probe. Once again, this only affects socket-related network events, for the same reason denoted above.

The solution requires making *Netto*'s eBPF layer scheduling-aware. This involves instrumenting the Linux scheduler, which we do by attaching a tracing program to the `sched_switch` tracepoint, that is invoked for each task swap. Coupled with the per-task storage introduced in the previous paragraph, this new program can correctly react to network task migrations. Similarly, the strategy is that of "impersonating" the associated event's entry and exit routines by the `sched_switch` probe based on the values of the task bits for the previous and next tasks.

2) Cost breakdown for the ***`NET_RX_SOFTIRQ`****:* The algorithm, as described to this point, is able to capture the total CPU utilization of the Linux network stack, but provides little to no insight into what components make up this monolithic cost. In fact, the only subdivision that can be attempted at this point is attributing the cost of the `NET_RX_SOFTIRQ` and `sock_recvmsg` to inbound traffic, and that of the remaining events to outbound, locally generated traffic. This classification is coarse and provides little use as a diagnostic data point; undoubtedly, a finer-grained breakdown could prove convenient in identifying hotspots and implementing optimizations. For this purpose, a `NET_RX_SOFTIRQ` breakdown feature has been implemented in *Netto*, which targets the receive-side of the network stack to extract usage metrics that reflect low-level network functions such as bridging, forwarding, and more.

Intuitively, the feature can be implemented by merely extending the set of kernel functions tracked with eBPF probes to also include the most significant from within the specific softirq handler, some of which are briefly described:

- **`br_handle_frame()`**: Bridge entry point, it is called as a *receive handler* for all Ethernet frames received on interfaces that are configured as part of a software bridge.
- **`do_xdp_generic()`**: Run any attached Generic XDP programs. Note that the native variant can not be tracked with this method, as the kernel function responsible for this activity is decorated as `inline` for performance reasons, and thus not traceable with eBPF.
- **`ip_forward()`**: Entry point for IPv4 forwarding of network packets.
- **`ip_local_deliver()`**: Deliver IPv4 skbs to upper layer protocols, eventually reaching sockets.

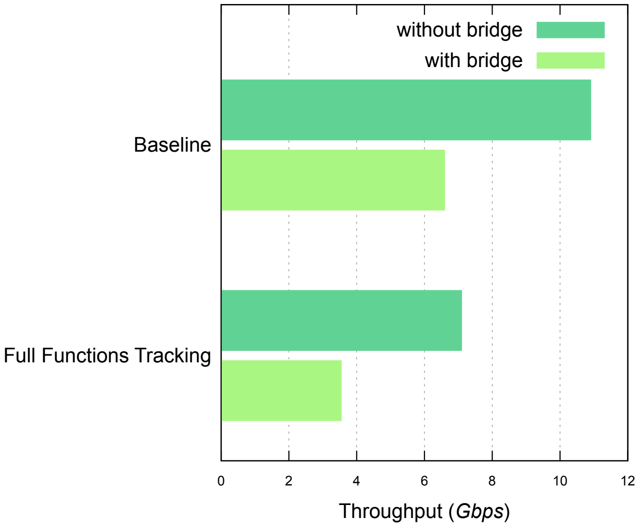

Fig. 2: Overhead of Full Functions Tracking on the iperf3 TCP receive test under different networking configurations. When bridging is enabled on the host effective speed is further reduced.

B. Network Stack Sampling

As an alternative to using eBPF to hook into the network stack functions directly, we also explored a sampling-based profiling solution, which would gather equivalent metrics by statistically evaluating arrays of captured stack traces.

The advantage of this methodology, compared to the one presented in the previous sub-section, is in the overhead associated with the measurement stack. The execution of eBPF tracing programs comes with a small computational overhead that is typically acceptable for most applications. However, *Netto* requires attaching to routines in the data path of the network stack which can have a very high invocation rate: for example, most of the programs used for the `NET_RX_SOFTIRQ` breakdown are executed for every incoming packet, and thus their instrumentation cost scales with the inbound throughput. This means that, in high speed networks, *Netto* can have a significant impact on the system's networking performance. Figure 2 shows an almost 50% reduction to TCP inbound throughput on a 40 Gbps link when GRO is disabled.

Conversely, a loss in accuracy is expected due to the sampling sparsity, although the magnitude of the error can be chosen to be as low as needed by controlling the sampling frequency setting.

The *network stack sampling* strategy implements a sampling profiler that specifically targets the Linux kernel's network stack. It works by instrumenting a purposefully allocated *perf event* that is configured to run periodically with a user-chosen frequency; for each raised perf interrupt, an eBPF program is run to capture the current CPU's kernel-side stack trace with the `bpf_get_stack()` helper. The traces are then delivered to *Netto*'s user-space controller for analysis and metrics extraction. The eBPF layer is hence significantly simplified with respect to the previous methodology, as much of the measurement's responsibility is moved to the user-space domain, contributing to a great reduction in system overhead.

Special care has to be taken for the traces transport solution in order to minimize additional and unnecessary costs: each stack trace can be up to 1 kB in size under default kernel configuration; this figure is then multiplied by the sampling frequency and system's core count to give the necessary stack trace bandwidth. Considering a reasonable sampling frequency of 1 kHz, this value easily reaches several MBps. For this reason, regular `BPF_MAP_TYPE_STACK_TRACE` maps are not suitable due to their lack of support for batch extraction, requiring the execution of multiple system calls for every pulled stack trace. Instead, *Netto* uses a `BPF_MAP_TYPE_ARRAY` map decorated with the `BPF_F_MMAPABLE` flag, which behaves similarly to raw mapped memory shared between the two actors. A double-buffering strategy is then employed to make sure that no traces are lost during the map-draining phase: the map is logically split into two equally sized slots; at any given point in time, exactly one of them is enabled for writing by the eBPF layer, and the other for reading by the user space controller. The roles are atomically swapped just before the controller begins draining the traces, ensuring that its view of the buffer remains stable during the entire operation.

Compared to an idiomatic `BPF_MAP_TYPE_RINGBUF` — which also internally uses a shared memory buffer between kernel and user-space — our solution shows a marginal but consistent improvement in trace extraction overhead, as depicted in Figure 3. Trace insertion was instead found to perform largely equally between the two map choices.

As the traces are received by the user-space controller, they are iterated over to match each stack frame to a set of predetermined symbols retrieved from the running kernel through the `/proc/kallsyms` virtual file. Matches for each symbol are accumulated and scaled by the sampling period in order to obtain their estimated on-CPU time.

V. Evaluation

In this section the proposed solutions will be evaluated and compared with regard to runtime performance and measurement accuracy. Then, multiple examples of real measurements from our testbed are presented to showcase the capabilities of the method, as well as providing a first analysis of the typical network stack costs under common workloads. Our testbed consists of two identical *Intel Core i7 6700*-powered machines (4 *Skylake* cores at 3.4 GHz base frequency, SMT enabled) running Linux 5.15 LTS and directly connected at 40 Gbps by a pair of *Intel XL710* QSFP+ NICs.

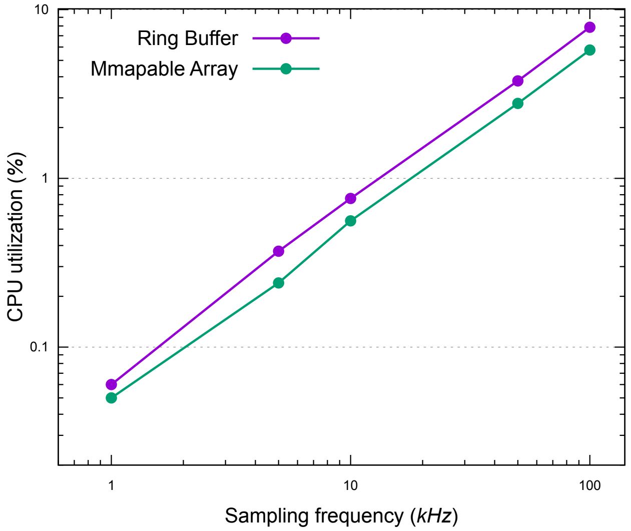

Fig. 3: CPU utilization for network stack sampling with ring buffer and `mmapable` array measured at various sampling frequencies.

A. Processing overhead

As mentioned previously, the *Network Stack Sampling* technique has been introduced with the intention of alleviating the severe performance concerns that *Full Functions Tracking* has been shown to suffers from in high packet rate conditions (Figure 2). To prove that the achieved level of overhead is in fact reduced and no longer debilitating to the overall system's performance, we subjected both techniques to a workload aimed at exposing their computational overhead.

The *iperf3* open source throughput measuring software was used to generate UDP traffic between two network namespaces of a Linux 5.15 host at a fixed rate of 1.5 Gbps. Artificially limiting the throughput allowed us to disregard possible divergences in the attained speeds, thus restricting any discrepancy in efficiency between the various methods to the amount of CPU used during the transfers. We then collected the one-minute average of the overall CPU usage for the cores involved in packet processing, and reported it in Figure 4.

The baseline number represents the ideal performance target, as it was obtained with no external instrumentation; impressively, *Network Stack Sampling* with a 1 kHz sampling frequency is able to match it, displaying a virtually null overhead. Predictably, as the frequency is increased, its cost also rapidly rises, but we argue that settings in excess of 1 kHz are generally not required to achieve a satisfactory level of accuracy. Meanwhile, *Full Functions Tracking* ("*FFT full*") is only able to barely outperform *NSS* when configured with the excessive 50 kHz setting, although a more acceptable overhead can be obtained by disabling its `NET_RX_SOFTIRQ` breakdown functionality (a feature which is always implicitly

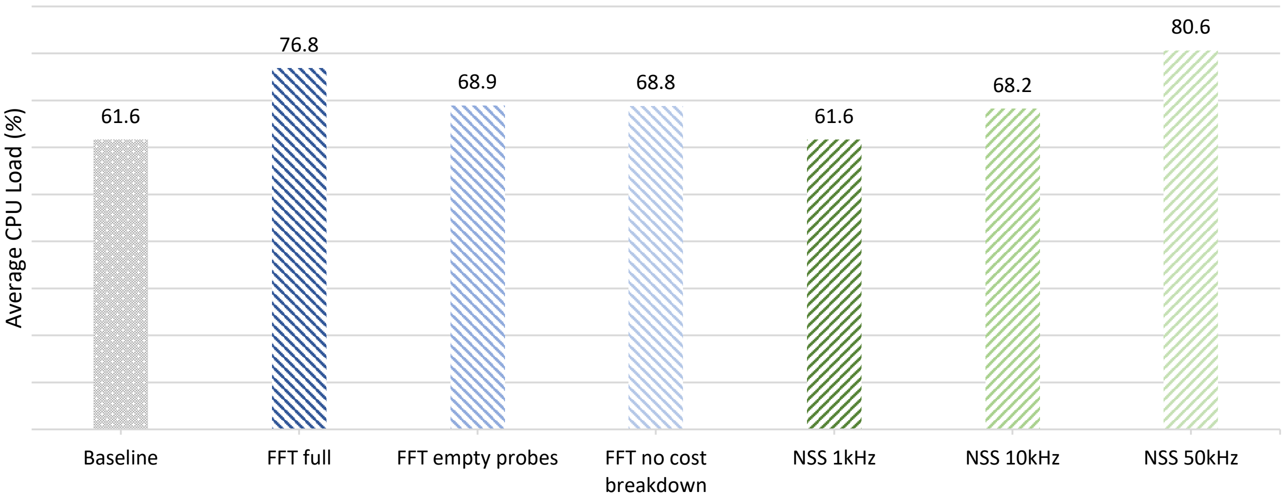

Fig. 4: Average CPU load measured during an iperf3 UDP transfer between two bridged network namespaces. The graph compares the baseline with no real-time network stack instrumentation (*gray*) to Full Functions Tracking (“*FFT*”, *blue*) and Network Stack Sampling (“*NSS*”, *green*).

enabled for *NSS*, as it comes at no extra cost). Finally, the “*FFT empty probes*” bar shows the overhead of *Full Functions Tracking* when its eBPF probes are attached but emptied out of any instruction; the fact that the overhead remains significant indicates that the eBPF instrumentation cost plays a major role in determining FFT’s poor performance.

Additionally, it must be noted that the general purpose Linux *perf* [10] profiler was able to achieve a consistently lower runtime sampling cost than even *NSS* — presumably due to the fact that it can implement an equivalent function without relying on the eBPF VM. Going forward, it’s clear that *Netto* is better off depending on an industry-standard performance monitoring backend such as *perf* instead of providing its own, but for the sake of this paper, our eBPF-based implementation can effectively show the strenghts and weaknesses of a sampling solution.

B. Accuracy

Full Functions Tracking and *Network Stack Sampling* take two radically different routes for producing corresponding metrics, but while *FFT* technically outputs ground truth values by directly measuring the target functions with nanosecond precision (although it must be noted that, due to the performance impact highlighted above, its results might be somewhat skewed by the instrumentation itself), *NSS*’s accuracy deserves a more thorough discussion.

In principle, the time quantization determined by the sampling frequency, together with the user-space integration time setting (i.e. the controller invocation period, here kept at the default $500\,\mathrm{ms}$), directly establishes the precision that the technique can accomplish. For instance, a $1\,\mathrm{kHz}$ frequency paired with a $1\,\mathrm{s}$ user-space update period gives a theoretical resolution of 0.1% for each captured metric. In practice, the user might choose to reduce the integration time to smaller intervals to lower the memory footprint, which would proportionally coarsen the resolution of the output.

In Figure 5 we compare the consistency of the extracted metrics for the `sock_sendmsg()` function specifically during a consistent UDP transmission over multiple sampling frequency settings. Every $500\,\mathrm{ms}$, the controller produces an estimate for the consumption of every metric; if the workload stays consistent (as it does in this test), the output metrics should ideally also remain constant throughout the testing period. The graph in Figure 5 is thus meant to isolate the jitter in the output values that results from *NSS*’s finite sampling frequency.

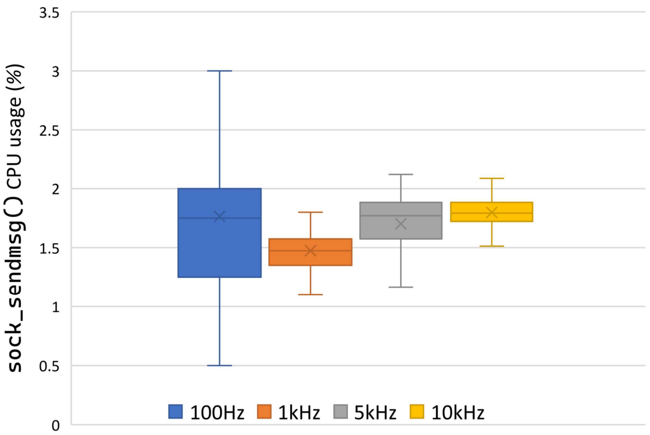

Fig. 5: Measurement stability of Network Stack Sampling at various sampling frequencies for the `sock_sendmsg()` metric under a consistent networking load.

At $100\,\mathrm{Hz}$, the sampling pattern is clearly too sparse to produce homogeneous results. As the sampling rate is increased, the variance improves, but after $1\,\mathrm{kHz}$ the gains remain limited. Also note that a slight inflation in the average metric’s value

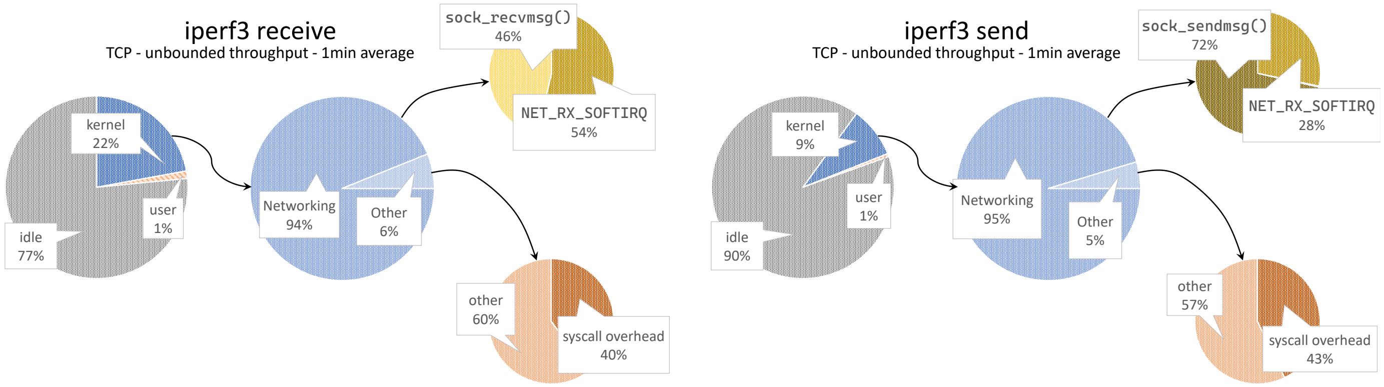

Fig. 6: Detailed CPU allocation breakdown for the iperf3 TCP receive (*left*) and send (*right*) tests. In both instances, the leftmost pie represents the overall distribution of the CPU among idle, user and kernel time; successive pies then progressively expand on specific sections.

is to be expected at higher sampling frequency settings, as the added load of the `perf` interrupt gets reflected in its own measurements; this is a further deterrent against needlessly using high sample rates.

C. Test case evaluation

To conclude the evaluation section, we now present some of the results that can be obtained with the methodologies described in this paper. In all cases, the numbers were collected on the setup described at the beginning of the section, and using *Network Stack Sampling* with a 1 kHz sampling frequency.

First of all, we analyzed the CPU load associated with simple synthetic network transfers via the *iperf3* throughput testing tool. Figure 6 shows two graphs depicting the receive and send workloads respectively, performed at line-rate TCP and averaged out over one minute of continuous testing. The graphs provide a progressively increasing level-of-detail and are meant to be read from left to right: the first pie shows the overall subdivision of the CPU time between idle, user and kernel modes; successive pies expand on the kernel time first, then its networking and "other" component (i.e. everything that can not be classified as networking).

For these basic workloads, most of the kernel time is spent within networking functions, such as the `NET_RX_SOFTIRQ`'s handler, as well as `sock_recvmsg()` and `sock_sendmsg()`. The rest of the privileged code being run is largely due to various forms of overhead, like that of system entry and exit. The most notable difference between the two transfer directions lies in the amount of recorded idle time, which indicates that the receiver side of the communication is the more resource intensive of the two. During these *iperf3* transfers, the *Normalized Networking Cost* — i.e. the percentage of CPU utilization exclusively attributable to the network stack, calculated disregarding the idle time — is respectively 88% and 90% for receive and send operations. This means that roughly nine out of ten clock cycles are due to networking instructions.

To represent a workload that more closely resembles that of production web servers, we tested *Netto* against Google's *Online Boutique* microservices demo [2] being hit with roughly 1000 reqs/s. The application implements a fake web store as a collection of collaborating microservices, which we deployed on top of a *KinD* [9] Kubernetes cluster. Figure 7 shows the collected measurements over a one minute period, once again using the *Network Stack Sampling* backend with a 1 kHz frequency.

This time, a sizeable portion of the total resources is allocated to user-space mode, presumably by the multiple microservices for processing requests and building responses. Additionally, only 17% of the kernel time can be confidently assigned to the networking domain; notably, an `io_uring SQPOLL` kernel thread takes up one entire CPU core by busy polling the ring's submission queue. Various other system calls then occupy most of the remaining registered kernel-mode time. Overall, the Online Boutique records a circa 8% *Normalized Networking Cost*.

VI. Discussion

Netto has been designed from its inception to only target in-kernel networking tasks. This covers the majority of the network-related work performed by typical servers up to and including the transport layer, but it fails to consider contributions from higher level protocols, as well as network technology that is implemented in user-space. This includes the QUIC reliable transport stack, user-space TLS, and kernel bypass custom data planes, along with application-level proxies that are becoming so common in the cloud-native world. Furthermore, *Netto* intentionally ignores NIC drivers' top halves, where the large variety of hardware vendors and products would impose unreasonable and continuous efforts in supporting all the possible configurations, though this aspect would hardly represent a significant portion of the overall CPU utilization.

In regard to future development, we believe the tool could be extended in several directions; first off, the event breakdown

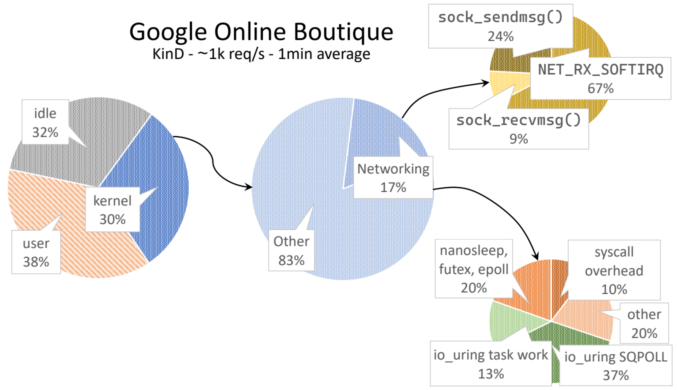

Fig. 7: Detailed CPU allocation breakdown for a host serving approximately 1000 requests every second of the *Google Online Boutique* microservices demo.

capability could be expanded to consider additional and more specific events that might be significant for some kinds of workload. Additionally, more work is required to give *Netto* a production-ready interface to show and log the captured metrics that can elegantly scale to data center sizes, as the current solution is based on a real-time *Grafana Pyroscope* dashboard, which offers no support for offline viewing of the sampled traces.

Furthermore, as mentioned in Section V-A, some extra performance can be gained for *Network Stack Sampling* by adopting *perf* as the sampling backend, thus making *Netto* a lightweight metrics filter and user interface with a networking-centric focus.

VII. Conclusions

In this paper we investigated the applicability of modern tracing technologies available in the Linux kernel (like eBPF) for achieving real-time instrumentation and visibility of the system's network stack. Specifically, we described two different paradigms that accomplish this goal: *Full Functions Tracking* and *Network Stack Sampling*. The former is based on the exact measurement of the execution time of each of the required kernel functions, while the latter approximates the same metrics by sampling the kernel's network stack and relying on its user-space component to perform the actual metrics extraction and implementing the measurement logic.

The subdivision of responsibilities that *Network Stack Sampling* provides is crucial to keeping the data path overhead low and minimizing the impact that the diagnostic solution causes to the overall system performance and responsiveness.

By contrast, *Full Functions Tracking* maintains the entire measurement stack inline with the network packet processing itself, causing slowdowns and inefficiencies. Additionally, hooking to network functions directly makes the eBPF probe invocation rate dependant on the host network activity, linking the measurement cost with the amount of packets processed by the system.

By applying these methods on a set of test workloads, we validated their results and assessed the effect on system performance, which — especially in the case of *Network Stack Sampling* — proved decidedly compatible with deploying as a background daemon in a production environment. Finally, we calculated a *Normalized Networking Cost* factor for all the above tests, which expresses the portion of the non-idle CPU time spent within the kernel's network stack, and found it to vary wildly between purely synthetic workloads ($\sim 90\%$) and realistic web server jobs ($\sim 8\%$).

References

[1] Amir Vahid Dastjerdi and Rajkumar Buyya. "Fog computing: Helping the Internet of Things realize its potential". In: *Computer* 49.8 (2016), pp. 112–116.

[2] Google. *"Online Boutique" microservices demo*. URL: https://github.com/GoogleCloudPlatform/microservices-demo.

[3] Roni Haecki et al. "How to diagnose nanosecond network latencies in rich end-host stacks". In: *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Renton, WA: USENIX Association, Apr. 2022, pp. 861–877. ISBN: 978-1-939133-27-4. URL: https://www.usenix.org/conference/nsdi22/presentation/haecki.

[4] *Intel®64 and IA-32 Architectures Software Developer's Manual*. 3C. Accessed: 2023-09-01. Intel Corporation. 2023. Chap. 33.

[5] Simon Peter et al. "Arrakis: The Operating System is the Control Plane". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 1–16. ISBN: 978-1-931971-16-4. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/peter.

[6] Luigi Rizzo. "netmap: A Novel Framework for Fast Packet I/O". In: *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association, June 2012, pp. 101–112. ISBN: 978-931971-93-5. URL: https://www.usenix.org/conference/atc12/technical-sessions/presentation/rizzo.

[7] Alexei Starovoitov. *BPF trampolines Linux patch*. Nov. 2019. URL: https://lore.kernel.org/bpf/20191114185720.1641606-5-ast@kernel.org/ (visited on 08/14/2023).

[8] The eBPF community. *eBPF*. 2023. URL: https://ebpf.io/ (visited on 07/17/2023).

[9] The Kubernetes Authors. *KinD: Kubernetes in Docker*. URL: https://kind.sigs.k8s.io/.

[10] The Linux Open-Source Authors. *Linux perf*. URL: https://perf.wiki.kernel.org/index.php/Main_Page.

[11] Wikipedia contributors. *io_uring - Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/wiki/Io_uring (visited on 08/22/2023).