POLITECNICO DI TORINO

Master of Science in Computer Engineering

Master Thesis

Assessing the Impact of Linux Networking on CPU Resources

Politecnico di Torino

Supervisor
prof. Fulvio Risso

Candidate
Davide Miola

October 2023

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 available in a data center. This work describes the design and functionality of a novel tool, *Netto*, that enables in depth observation and monitoring of the Linux kernel's networking stack in real time, by exploiting the tracing capabilities of eBPF, an affirmed technology which dramatically enhances Linux's observability by allowing the dynamic injection of user code into the kernel. It is also shown how a dynamic breakdown of the individual components of the measured networking cost can be built on the fly by collecting and analyzing CPU stack traces.

Contents

Chapter 1

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*) allowing networked communication, and (*ii*) minimizing their computational overhead on end hosts. Therefore, as network traffic continues to surge with the widespread adoption of IoT (Internet of Things) devices [2] 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 exchanged data.

To address this challenge, emerging technologies such as eBPF (extended Berkeley Packet Filter) [12] have gained significant attention due to their ability to perform dynamic and non-intrusive tracing of the whole Linux kernel. 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.

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.

1.1 Goal of the Thesis

The primary objective of this work is to explore the use of eBPF as a powerful tool for tracking and quantifying the CPU cost induced by the networking stack of the Linux kernel, and presenting *Netto* (`https://github.com/miolad/netto`), a novel tool that utilizes these principles to allow the real-time, low overhead monitoring and analysis of Linux's networking layer.

The following chapters will present a comprehensive methodology for utilizing eBPF to track kernel overhead in the networking stack, and discuss the instrumentation of critical network-related functions using eBPF probes, collection and analysis of performance data, and interpretation of the results. Then, the details of the current implementation of these methodology is examined, together with the challenges that have been hit throughout the writing of *Netto*. Finally, the tool has been applied to several real-world scenarios, to validate it and gauge its accuracy and performance overhead.

Chapter 2

Related Work

The demand for efficient and accurate measurements of the costs and overheads of the kernel's network stack is pervasive in all research work surrounding the networking topic. A reliable measurement is in fact the ideal starting point from which all research should begin, as well as the target, to ensure expected performance gains or rule out regressions.

Indeed, this work's subject is touched upon by much of the relevant literature, whether it is the main point of the paper, or a necessary note. In any case, regardless of this fact, no unified measurement framework exists that can satisfy all the usual requirements of accuracy and transparency, and in no case the suggested method proves exhaustive in measuring the entire networking layer of the Linux kernel, as *Netto* vouches.

In the following sections, three notable examples are reported to discuss advantages and limitations of the suggested techniques.

2.1 *Netmap*

As part of the *netmap* project, Rizzo [9, Section 2.3] developed a custom solution to measure the baseline performance profile of FreeBSD's `sendto` system call. In the adopted technique, the syscall was instrumented in-kernel to allow forcing an early return at different depths. The designated system call could then be profiled by calling it repeatedly from an ad-hoc user-space program and averaging out its execution time over multiple runs.

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.

2.2 *Arrakis*

Two years later, Peter et al. [8] used a similar technique to profile the UDP data path of the Linux network stack. Unlike [9], however, measurements were taken by timestamping meaningful events directly in kernel. 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 this complications by conditionally sampling these otherwise prohibitive functions.

2.3 *NSight*

NSight [5] 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

Chapter 3

Background

3.1 eBPF: Extended Berkeley Packet Filter

Extended Berkeley Packet Filter — or *eBPF* [12] — is a versatile and powerful technology that has gained prominence in recent years for its ability to address a wide range of networking and system monitoring challenges.

Initially introduced in the Linux kernel in version 3.18 as an extension of the traditional Berkeley Packet Filter (*BPF*, now referred to as *classic* BPF, or *cBPF*), eBPF has evolved far beyond the original scope of packet filtering for traffic capture, and it is now a robust framework for programmable packet processing, system tracing, and custom event handling for modern computer systems.

Central to its functionality is a Just-In-Time (JIT) recompiler, which allows users to write and execute programs directly within the kernel, providing an unprecedented level of flexibility and efficiency in network and system management.

Key features and components of eBPF include:

1. **Programmability**: eBPF programs are written in a restricted subset of the C programming language (sometimes referred to as *BPF-C*, although compilation from other languages *is possible*[1]), ensuring safety and security. These programs can be dynamically injected into the kernel and executed without requiring kernel module recompilation or system reboot, enabling rapid development and deployment of custom functionality.

2. **Security**: eBPF's safety features, including bound checking and instruction verification, are enforced thanks to a static verifier which is invoked upon program load. These guarantee the correctness of the injected code, making eBPF a secure choice for running untrusted code within the kernel. This is

[1] *Aya* (https://aya-rs.dev/book/) is an in development, end-to-end framework for utilizing eBPF with the Rust programming language.

particularly important in protecting the integrity and stability of the host system.

3. **Compatibility**: eBPF is a core feature of modern versions of the Linux kernel, meaning that support is confirmed regardless of the host Linux distribution. Moreover, the eBPF instruction set is independent of the underlying physical architecture, making eBPF programs inherently portable.

4. **Performance**: eBPF programs are executed directly in kernel, which eliminates the need for expensive system to user transitions, that often represent a significant performance pitfall of modern operating systems' design. Furthermore, thanks to its JIT recompiler, eBPF code can be run at near native speed.

5. **Ecosystem**: A rich ecosystem of tools and libraries has emerged around eBPF, including *BCC* (the *BPF Compiler Collection*, TODO: cite), which provides pre-built eBPF programs for common use cases, and *bpftool* for managing eBPF programs and maps at runtime.

6. **Versatility**: Thanks to the many in-kernel hook points available to its programs, eBPF is suitable for a multitude of different applications, ranging from the definition of fast network functions, to security monitoring, to system tracing, and beyond.

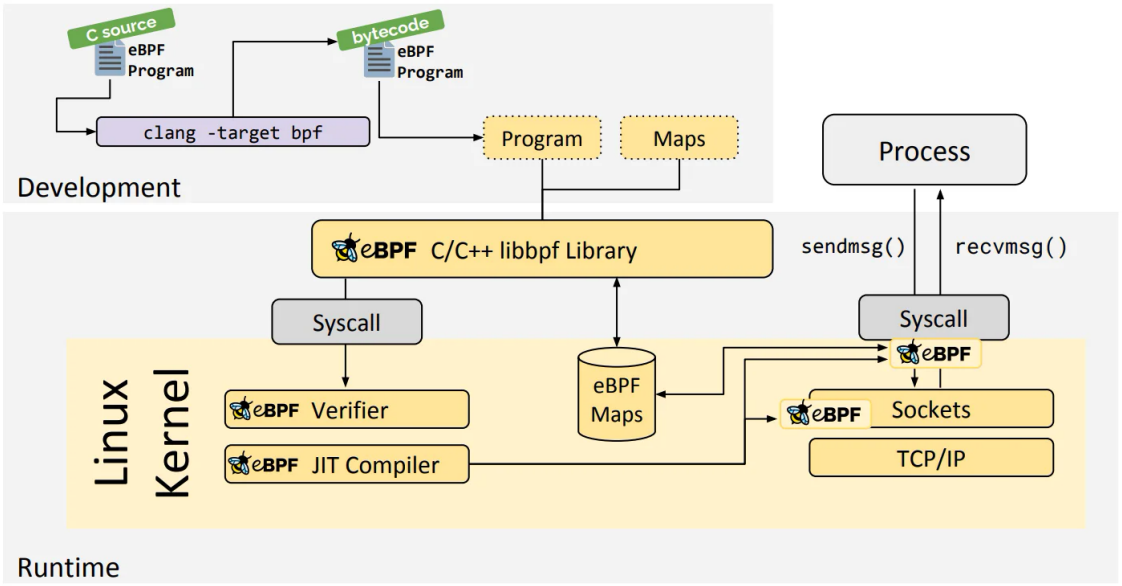


Figure 3.1: High level overview of the eBPF architecture.

The flexibility and performance benefits of eBPF have positioned it as a fundamental tool for enhancing the observability, security, and efficiency of modern computer systems.

Some of the most notable components of the eBPF architecture are discussed in the following sections.

3.1.1 Maps

The eBPF virtual CPU's memory subsystem can only rely on a 512 B stack; no heap is available, and thus dynamic memory allocation is not supported. This apparent oversight in the design of eBPF is actually one of the main pillars of its security model: dynamic memory is in fact one of the largest sources of runtime faults for user programs, due to the error-prone nature of writing memory allocation and management code. Additionally, bringing `malloc` to the kernel could also have negative performance implications.

Instead, eBPF relies on *maps* to persist state across program invocations and share it with other eBPF programs in the kernel or with user-space applications. Fundamentally, maps are key/value stores managed entirely by the Linux kernel. eBPF programs can access them with dedicated helper functions[2], whereas user-space applications can read and write to them by submitting the relevant command through the `bpf` system call.

Many different map types are available, which differ by internal structure and behavior, as well as the exposed interface. In the following, some among the most noteworthy map types are described:

- `BPF_MAP_TYPE_ARRAY`: A simple linear array, it is fully described by the number and size of its entries. When decorated with the `BPF_F_MMAPABLE` flag, Linux allows user-space applications to "`mmap`" the backing memory to their address space, enabling them to access it without expensive system calls. This is the most direct and low-level means of exchanging data between eBPF programs and user-space applications.

- `BPF_MAP_TYPE_HASH`: A typical hash-based associative table, which maps keys to values. Both key and value types can be either primitive or composite types for maximum flexibility. Like `BPF_MAP_TYPE_ARRAY`, hash maps are size bounded by a `max_entries` parameter that is specified at map creation time, but unlike arrays, pre-allocation of memory can be turned off in favor of an allocate-on-insert strategy.

[2]Helpers are functions available to eBPF programs whose implementation resides in the kernel. Helpers allow to expose a stable API to the eBPF layer regardless of the underlying kernel version, and serve as bridge between the eBPF Virtual Machine and the native resources of the host system.

- `BPF_MAP_TYPE_PERF_EVENT_ARRAY`: Supports streaming data from eBPF to the user-space application by pushing raw bytes to a collection of per-CPU circular buffers.
- `BPF_MAP_TYPE_RINGBUF`: Newer and better alternative to the aforementioned `BPF_MAP_TYPE_PERF_EVENT_ARRAY`, to which it should be preferred in all cases. It overcomes some of the shortcomings of its predecessor, making it universally more convenient and/or performant. In a later chapter, this map type will be compared to the previous *mmapable* array for fast kernel-to-user data transfers.
- `BPF_MAP_TYPE_STACK_TRACE`: Complements eBPF's capability to dump CPU stack traces by providing a natural map to store and later retrieve them from the user-space. Conceptually, it functions as a hash map that associates a *stackid* to each inserted stack trace, with the former being an automatically generated 32 bit digest of the associated trace. Because traces that hash into the same *stackid* are not repeated, this map can provide a form of "*in-kernel trace summarization*", minimizing the number of traces that must reach the user-space controller.

Many more map types are available for different use cases. Additionally, some of the aforementioned maps are also provided in different variants, like the `PER_CPU` modifier available for arrays and hash maps (which partitions them into isolated, per-CPU copies), or the `LRU` alternative of hash maps (that adds *Least Recently Used* semantics to the map entries).

3.1.2 Program Types

eBPF programs are dynamically injected into the Linux kernel at specific *hook points*, that determine the circumstances and context with which the program is invoked. A *hook point* is defined by the tuple of *program type* and possibly a specific attachment point: the former determines the kind of program and set of concrete attachment points available, if any.

Among the multiple program types defined in modern versions of Linux, the tracing and monitoring possibilities are virtually limitless. Here are some of the most relevant eBPF program types for this work:

- **Tracepoints**: Linux tracepoints are static tracing hooks scattered all around the kernel's source code. A tracepoint is defined in the code by its name and context, i.e. what data is passed to tracing programs and with what format.

  Tracepoints identify fixed positions in the kernel source, generally associated with meaningful events; they are also organized into major categories — like `syscalls` or `irq` — that group related tracepoints together (see Listing 3.1 for reference).

eBPF supports attaching to tracepoint hooks through a few different program types, including (*i*) `tracepoint`, (*ii*) `raw_tracepoint`, and (*iii*) `tp_btf`. (*ii*) and (*iii*) differ from (*i*) because they're *raw*, meaning that tracepoint arguments are passed as raw bytes instead of parsing them into distinct variables beforehand (giving them a slight performance advantage); additionally, `tp_btf` programs are *BTF-powered*[3], thus allowing them to access the kernel's memory directly without bridging helpers.

- **KProbes**: Although tracepoints are numerous and well distributed in the kernel source, sometimes it is necessary to monitor functionality that is not covered by any static tracepoint. KProbes are effectively dynamic hook points that can be created at runtime to target almost any instruction of the running kernel. This is accomplished by replacing the live instruction from system memory at probe registration time with a breakpoint that traps the CPU and jumps to user code.

  Despite the convenience, these probes are generally very expensive.

- **`fentry/fexit`**: Since Linux 5.5, *BPF trampolines* [11] have allowed kernel code to call into eBPF programs directly and with virtually zero overhead. `fentry` and `fexit` are the two eBPF tracing program types built on top of this feature, which ought to replace the older KProbes thanks to their much superior performance.

  They can attach to, respectively, any kernel function's entry and return addresses and, similarly to the previously cited `tp_btf`, `fentry` and `fexit` are also powered by BTF, simplifying their usage with respect to program arguments and access to the host system's memory.

  Due to their undisputed superiority, *Netto* favors BFP trampolines to KProbes whenever tracepoints are not available.

- **`perf_event`**: `perf` is a powerful profiling tool included in the Linux kernel; it can instrument CPU performance counters and much more. eBPF programs of type `BPF_PROG_TYPE_PERF_EVENT` can be attached to an open `perf_event`, hence getting triggered whenever the event's counter is ticked.

  This is a very powerful feature that is exploited by *Netto* to achieve a periodically invoked eBPF program, that it uses to sample the CPU call stack at regular intervals.

[3]BTF (*BPF Type Format*) [14] is the metadata format which encodes the debug info related to eBPF programs and maps. When available, it can enhance program introspection and visibility, improving their portability.

```
$ ls /sys/kernel/tracing/events

alarmtimer          hda_controller  maple_tree      rtc
amd_cpu             hda_intel       mce             sched
asoc                header_event    mctp            scsi
avc                 header_page     mdio            sd
block               huge_memory     mei             signal
bpf_test_run        hwmon           migrate         skb
bpf_trace           hyperv          mmap            smbus
bridge              i2c             mmap_lock       snd_pcm
cfg80211            i2c_slave       module          sock
cgroup              i915            mptcp           sof
clk                 initcall        msr             sof_intel
cma                 intel_iommu     napi            spi
compaction          intel-sst       neigh           swiotlb
context_tracking    interconnect    net             syscalls
cpuhp               iocost          netlink         task
cros_ec             iomap           nmi             tcp
csd                 iommu           notifier        thermal
damon               io_uring        nvme            thermal_power_allocator
dev                 ipi             oom             thp
devfreq             irq             osnoise         timer
devlink             irq_matrix      page_isolation  tlb
dma_fence           irq_vectors     pagemap         ucsi
drm                 iwlwifi         page_pool       udp
enable              iwlwifi_data    percpu          v4l2
error_report        iwlwifi_io      power           vb2
exceptions          iwlwifi_msg     printk          vmalloc
ext4                iwlwifi_ucode   pwm             vmscan
fib                 jbd2            qdisc           vsyscall
fib6                kmem            ras             watchdog
filelock            ksm             raw_syscalls    wbt
filemap             kvm             rcu             workqueue
fs_dax              kvmmmu          regmap          writeback
ftrace              kyber           regulator       x86_fpu
gpio                libata          resctrl         xdp
handshake           lock            rpm             xen
hda                 mac80211        rseq            xhci-hcd
```

Listing 3.1: Available tracepoint *categories* as of Linux 6.5.2

3.1.3 Toolchain

As observed previously, eBPF programs are usually written in a restricted C language, then compiled by an LLVM-powered compiler — like *clang* — into the BPF bytecode, which can then be loaded into the kernel.

Most eBPF projects (like *Netto*) are composed of one or more eBPF programs

running in the kernel and a user-space controller that supervises them. To aid developers in adding eBPF capabilities to their tools, various toolkits and libraries are available on the open-source landscape. Among these, *libbpf* — a C library — is the most low-level option, as it only serves as a thin wrapper on top of the Linux `bpf` system call. It offers helpful APIs for loading compiled eBPF programs and maps, as well as attaching the verified programs to the chosen kernel's hook points. Due to *libbpf* being developed alongside the main Linux kernel, it is always in feature parity with mainline eBPF, and bindings for a number of other programming languages are available: *Netto* is built on top of the Rust binding of *libbpf*.

Another very popular option is *BCC*, the *Bpf Compiler Collection*. *BCC* is an assortment of tools and ready-made eBPF programs that aims to ease eBPF development by providing a Python API for compiling and interacting with eBPF code. Contrary to *libbpf*, *BCC* does not support the *CO-RE* programming paradigm (*Compile Once-Run Everywhere*): tools based on *BCC* must ship their eBPF code as source, for it to be compiled only at runtime. Because of this, it is also required of target systems to install the *BCC* suite in order to run the programs, which is one of the main reasons why *Netto* is not built on *BCC*.

3.2 RAPL: Running Average Power Limit

Among *Netto*'s features is the capability to estimate the host system's power draw owing to only networking-related tasks. This is crucial information necessary — for instance — to assess whether a SmartNIC deployment would be a sensible option for energy concerns in a data-center context. In order to give *Netto* energy awareness, the RAPL [6, Volume 3B, Chapter 15.10.1] interface was used.

RAPL — which stands for Running Average Power Limit — is a technology developed by Intel to monitor and control the power consumption in computer processors. Although first introduced for Intel chips, RAPL is now available on AMD CPUs as well.

RAPL consists of an MSR-based interface through which energy metrics about the running system are exposed, along with a method for setting power limits (for example to reduce power consumption or limit the amount of generated heat), though this aspect of the RAPL specification has not been used by *Netto*.

RAPL implements four distinct power domains that can be independently configured and probed (Figure 3.2 depicts them on a schematic picture):

1. **Package domain**: This includes all energy used by the CPU package, which contains both *PP0* and *PP1* subdomains.

2. **DRAM domain**: Represents the connected DRAM memory modules.

3. **PP0/Core domain**: Concerns all the cores of the processor.

4. **PP1/Graphics domain**: For CPU models that have it, this domain maps to the on-board GPU.

Because RAPL's resolution is limited to the entire core cluster of a CPU, *Netto* can not estimate the power draw for the individual core or hyperthread, and hence the estimated networking energy consumption can only be a rough approximation based on a linear correlation between CPU utilization and energy consumption.

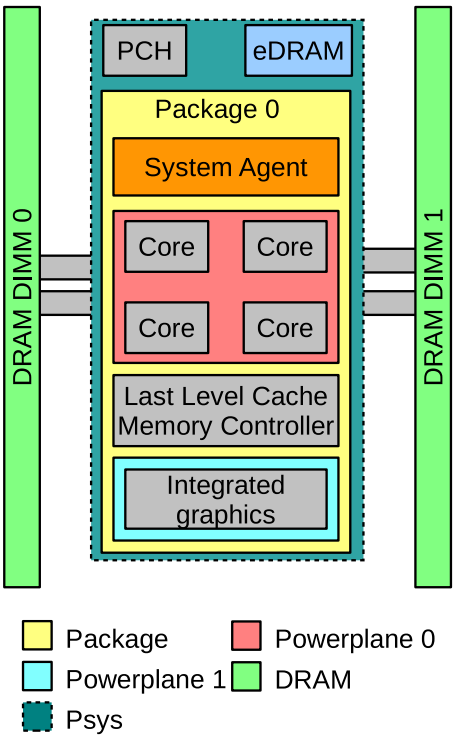


Figure 3.2: Power domains supported by RAPL.

3.3 Linux Networking Summary

The Linux network stack is the portion of the kernel that is in charge of handling and moving network packets, between network interface and application (and vice-versa), and from a network interface to another (for bridging and forwarding), regardless of whether interfaces are physical NICs or virtual devices that live entirely in software.

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

3.3.1 Network to socket

When a packet is received on a physical interface, it is DMAed to a ring 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.

Linux deferrable functions enable one to schedule code to be executed at a later time, thus allowing to bring expensive computations out of the constrained interrupt context; this makes them ideal for implementing typical driver's bottom halves. In modern versions of the Linux kernel, several facilities implement this abstraction, some of which are now briefly described:

- **Softirqs**: Softirqs are ideal for high frequency invocations. They are statically allocated, hence not applicable for implementing generic drivers' bottom halves; as of Linux 6.5.3, ten softirq variants are defined.

  Softirqs execute independently on different cores of the hosting computer, allowing true parallel processing. Each CPU has a bitmask of pending softirqs, that identifies which softirq types have been scheduled for execution with the `raise_softirq()` function. Regularly, each CPU checks for pending softirqs and serves them with `__do_softirq()`; most notably, this check happens at every IRQ exit. Because softirqs might get reactivated while being served (for example by high priority interrupts, or by the softirqs themselves), the `__do_softirq()` function is allowed to drain the pending mask up to ten times per invocation. Past this threshold, outstanding softirqs are moved to the process context via the *ksoftirqd* kernel threads, which call `__do_softirq()` in a loop. Given that any individual softirq handler is run with preemption disabled, their execution time must remain bounded, in order to avoid starving other tasks of the CPU.

- **Tasklets**: The preferable method to implement bottom halves for most situations is with tasklets. Tasklets are built on top of two specific softirqs (`HI_SOFTIRQ` for high priority tasklets, and `TASKLET_SOFTIRQ` for normal ones), but unlike softirqs, tasklets can be allocated and initialized at runtime, and they are serialized on different CPUs, meaning that tasklets of the same type are not allowed to run in parallel, simplifying the design of driver stacks by denying most sources of race conditions.

- **Workqueues**: Workqueues are similar to tasklets, in that they allow user allocation and management of deferred work units. Conversely, though, the implementation of workqueues is based on a worker pool made up of kernel thread

workers, which host the deferred computation. For this reason, workqueues are always run in process context, and can therefore use blocking calls.

- **Threaded IRQs**: Threaded interrupt handlers have been a feature of the Linux kernel since 2.6, with the introduction of `request_threaded_irq()`. In contrast to the traditional `request_irq()`, it not only allows adding a handler for an interrupt line, but it also registers an associated `thread_fn`: after each interruption from the specified irq, the regular handler is first executed in interrupt context, then the system will independently wake up a dedicated kernel thread where the specified deferred function will later be run.

Due to the networking layer being a fundamental component of a modern operating system, and considering the possibly high frequency of exchanged network packets, which would require the recurrent invocation of the kernel's network stack, softirqs are the most well suited option for its implementation. Indeed, two of the ten currently allocated softirq types are related to networking: `NET_RX_SOFTIRQ` and `NET_TX_SOFTIRQ`.

For NIC drivers implementing the NAPI[4] interface, the top half culminates with a call to `napi_schedule()`, which raises the `NET_RX_SOFTIRQ` on the local CPU and registers the NIC's driver for polling. 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 rings: a process which can be broken down into the following high level overview, for each received packet:

1. Run any `XDP_NATIVE` eBPF program on the packet, 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 generic networking layer of the Linux kernel is reached, an 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()`. Figure 3.3 shows a simplified model of the insides of the `NET_RX_SOFTIRQ`.

[4]NAPI [15] (formerly *New API*) is the event handling mechanism used in the Linux networking stack.

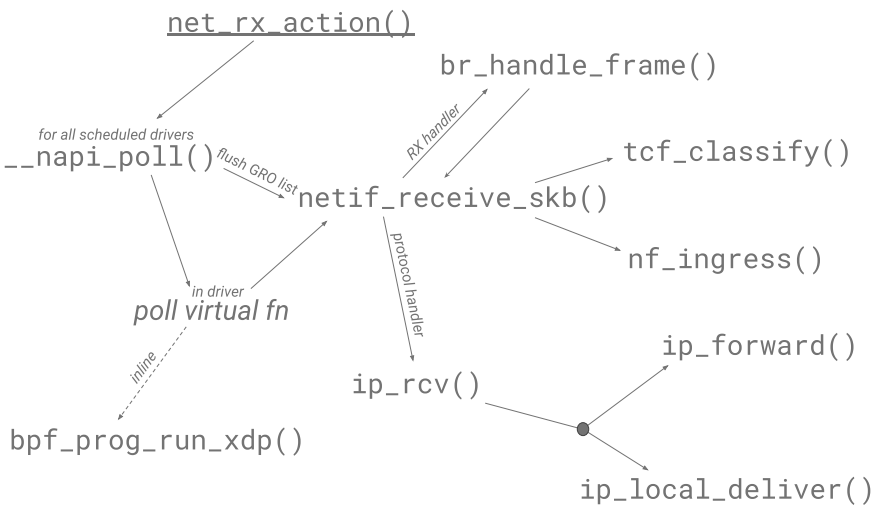


Figure 3.3: Simplified model for the `NET_RX_SOFTIRQ`; the entry point is `net_rx_action`, registered as the handler for this softirq.

3.3.2 Socket to application

To provide networking services to user-space 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, 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` will 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, but none have had the success and prominence of the recent ***io_uring*** [18] interface.

`io_uring`

The Linux kernel has provided support for asynchronous input/output (*AIO*) operations since version 2.5, but the implementation has largely been perceived as inefficient and limited, with glaring oversights such as the complete lack of support for socket operations. With Linux 5.1, `io_uring` has officially replaced the obsolete

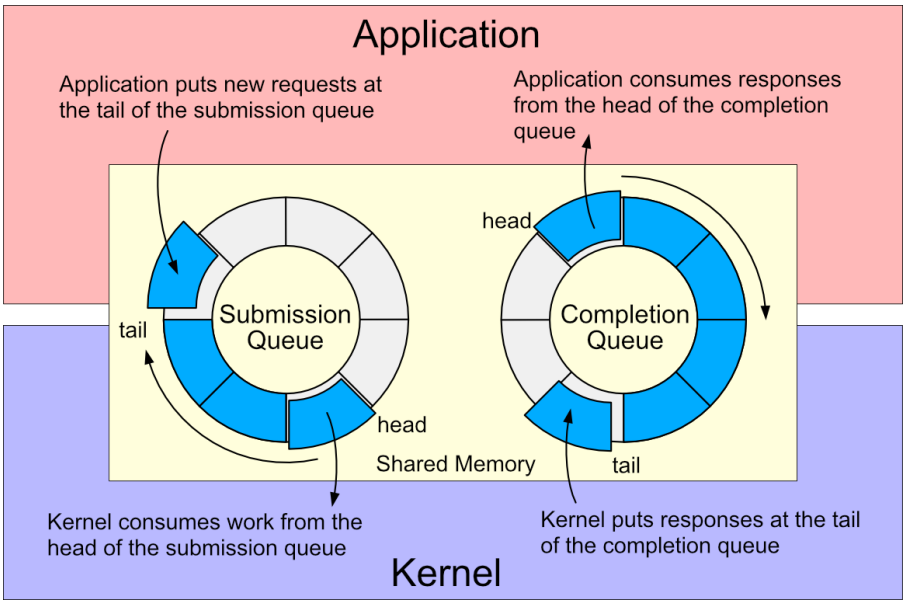


Figure 3.4: Design of the `io_uring` asynchronous I/O interface.

AIO as the preferred asynchronous I/O interface.

`io_uring` is a high performance asynchronous I/O framework proposed as an alternative to the traditional syscall interface, and intended to simplify and speed up asynchronous operations by minimizing context switches and reducing system call overhead.

It is based on two circular buffers in shared memory between client application and kernel, respectively called "*submission queue*" (SQ) and "*completion queue*" (CQ); the former being writable by the user and readable by the kernel, vice-versa the latter is writable by the kernel and readable by the user. Keeping these rings in mapped memory allows to negate the cost of expensive system calls for writing or reading them by user code.

An I/O request is issued by pushing the proper entry to the submission queue (SQE, or Submission Queue Entry), describing the desired operation. The kernel notices the submission either by direct notification from the user with the `io_uring_enter` syscall, or through a dedicated polling thread. The latter option (known as `SQPOLL`) allows for true zero-syscall I/O, at the expense of burning a CPU core on polling. In any case, as the kernel receives an SQE, the associated operation is asynchronously executed. Upon termination, its return code is wrapped in a CQE (Completion Queue Entry) and written to the CQ, where the application can receive it, completing the cycle. Figure 3.4 depicts this mechanism.

3.3.3 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 neutral interrupts and softirqs, 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 contained within the syscall, which brings data from the user-space buffer to the output NIC driver. 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 to aid its congestion and flow control implementations.

Lastly, the previously mentioned `NET_TX_SOFTIRQ` is infrequently raised to flush NICs transmit queues in case an interface data is to be transmitted on was busy on previous send attempts.

Chapter 4

Netto's Architecture

This chapter will present *Netto*'s architecture and design choices, first as a high level overview of the methodology behind its main monitoring capabilities, then by discussing some of the necessary precautions against scheduling and task interruptions. Finally, *Netto*'s event breakdown feature is going to be analyzed in its multiple proposed forms.

4.1 Overview

In order to accurately measure the CPU utilization of the Linux kernel's networking stack, *Netto* uses eBPF tracing 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, compatibility is ensured 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 have thus been identified — which will be referred to as "*events*" from now on —, together with their associated C functions in the kernel source code are the following:

- `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 *skbuff*s per invocation by default.

  This softirq is responsible for most of the receive-side network stack and *reactive* outbound traffic (i.e. transmitted packets that originate from remote hosts, such as for bridging and forwarding network functions). Other kinds of protocol-specific output packets are also produced in this softirq's context, like TCP acknowledgements or ICMP echo replies.

- `NET_TX_SOFTIRQ`: Occasionally flushes transmission queues for busy networking drivers during high load scenarios.

Despite the name similarity with the `NET_RX_SOFTIRQ`, this softirq can not be considered as its transmit-side sibling. In fact, its CPU footprint is generally quite limited.

- **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 on data through a socket object. Unlike socket reads, 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 4.1 summarizes the chosen eBPF program types and attach points for the four aforementioned events. Note that eBPF ELF sections[1] are specified in the format `<program type>/[<attach point>]`. For the first two events, which use BTF-powered raw tracepoint program types, the attach point refers to the specific tracepoint's name used, whereas for the latter events, the trampoline based tracing programs `fentry` and `fexit` require as their attach point the name of the kernel function to hook.

The decisions about eBPF program types is attributable to an effort at minimizing in-kernel overhead by the eBPF instrumentation. For this reason, static tracepoints were preferred whenever available, and BPF trampolines were selected otherwise. Note that, as both softirq events map to the same set of tracepoints, the two are distinguished by filtering over their `vec` argument. Also worth mentioning is that the last two events are hit by any of the available relevant system calls, including their async 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* eBPF 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 via a suitable `BPF_MAP_TYPE_PERCPU_ARRAY`

[1] 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.

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`

Table 4.1: eBPF programs' ELF sections for the four identified networking entry-points.

eBPF map. The controller runs with a default period of 500 ms, reads the most up-to-date values from the map — on which it performs the necessary integration analysis by comparing them with the previous iteration's counters —, and then updates the user-facing report. In particular, the total amount of CPU utilization spent by Linux in networking tasks during any controller update cycle is given by the fraction of the sum of every event's on-CPU time since the last controller invocation over the controller period, for each CPU. The default control loop invocation frequency of 2 Hz has been originally chosen as a good compromise between a satisfying temporal resolution of the output data and an acceptable level of CPU overhead, although it must be noted that the initial concerns about elevated system load due to the user-space controller have not materialized (as is shown in Chapter 6). In the current implementation of *Netto* the default value can thus be overridden to best suit the user's needs.

To provide more meaningful insights into the cost of networking, the `NET_RX_SOFTIRQ`'s contribution to the measurement is then broken down into smaller components. This allows *Netto* to estimate the cost of the individual network functions like bridging and forwarding. A more in depth explanation of this feature, along with details on the design of two competing methodologies that have been attempted for its implementation, is provided in Section 4.3.

4.2 Handling Switching of the Execution Context

The simple strategy presented so far works well in the common case, but it is subject to errors in preemptible kernels in two separate scenarios: (*i*) network softirqs *can* interrupt tasks on which socket events are running (this is because softirqs could be run in interrupt context), and (*ii*) the task scheduler could preempt or migrate monitored tasks; together, these synchronization problems aggravate the eBPF side of the measurement stack.

The following subsections are going to present the proposed solutions to both of these complications, discussing the resulting architecture.

4.2.1 Softirqs Interrupting Other Tasks

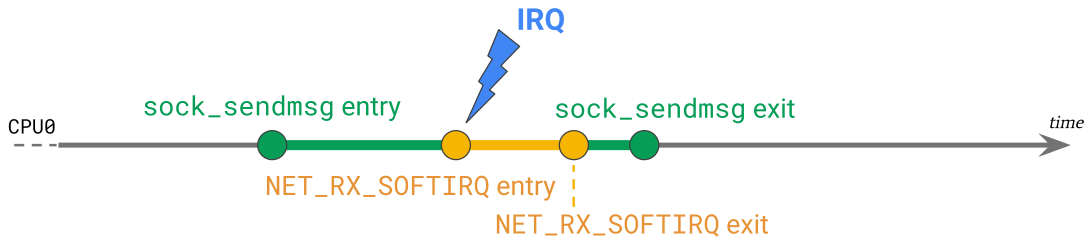


Figure 4.1: Diagram of a softirq interrupting another networking task (a `sock_sendmsg` in the figure).

Linux softirqs can be run in irq context, and can thus interrupt any user process, including those where an instance of the identified socket network events is currently running. This occurrence (schematically depicted in Figure 4.1) is especially common during high network load situations. This unfortunate timing of events would lead — under the logic explained in the previous section — 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 under the Linux implementation of softirqs, because they are run in a non-preemptible context.

The adopted solution consists in associating to each encountered kernel task[2] (i.e. each task a socket operation is found running on) a pair of bits allocated to store information about what socket operation is currently executing, if any. It's important to associate this data to the task, and not to the host CPU, because tasks can be migrated between cores, and the information must be maintained even for threads that have been scheduled out of the processor.

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.

4.2.2 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, as shown in the schema in Figure 4.2. Since metrics exported by the eBPF layer have a per-CPU resolution,

[2]In Linux, a *task* represents a thread; each task has a *pid* that identifies it. Multi-threaded processes are instead identified by a *tgid* (*Thread Group ID*).

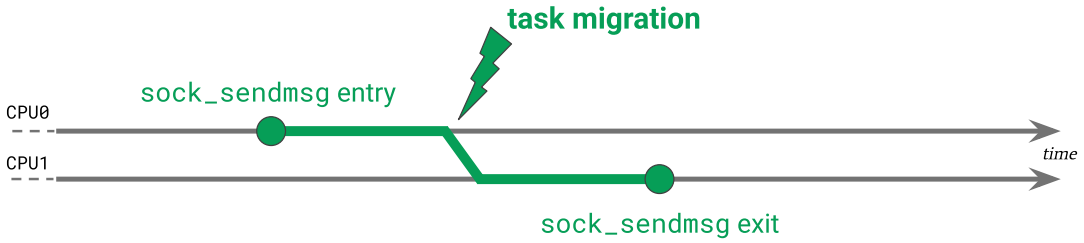


Figure 4.2: A task being migrated between two CPUs of the underlying computer system.

any such migration negatively influences the measurement's accuracy, as the exact on-CPU time of a migrated task can not be correctly estimated by its exit eBPF probe. Again, this problem only affects socket-related network events, for the same reasons denoted above.

The solution here involves making *Netto*'s eBPF layer scheduling aware. This involves instrumenting the Linux scheduler, which can be done by attaching a tracing program to the `sched_switch` tracepoint. This tracepoint, in fact, is invoked at every task switch on every CPU, and can access information of the entering and exiting tasks. Coupled with the per-task storage introduced in 4.2.1, this new program can correctly react to network task migrations. Once again, the elected strategy consists in 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.

4.3 Event Breakdown

The tool, as described thus far, is fully functional in presenting the total CPU utilization of the Linux networking stack, but provides little to no insight into what components make up this monolithic cost. In fact, the only subdivision we can attempt at this point is the one that spreads the overall CPU load into the four identified events, though the diagnostic advantage of this operation would be limited. It is instead undeniable that a finer-grained breakdown of the total cost could prove useful in identifying hotspots and implementing optimizations.

For this reason, one of the key capabilities developed for *Netto* as part of this thesis is providing an estimated breakdown of the cost of the individual network events into their main components. With this feature — currently implemented for the `NET_RX_SOFTIRQ` event — *Netto* can approximate the cost of the individual network functions of a Linux host, such as bridging and forwarding.

In the remaining subsections of this chapter, two techniques designed to implement this goal are first going to be introduced, *Full Functions Tracking* and *Network Stack Sampling*. Chapter 5 is then going to detail their implementation and expose complications associated with their realization. Finally, Chapter 6 will compare their performance and overhead, revealing the preferred technique used in upstream *Netto*.

4.3.1 Full Functions Tracking

The most intuitive solution would be to extend the set of events traced with eBPF from the initial four, with the core functions found in the body of the `NET_RX_SOFTIRQ` and highlighted in the simplified model in Figure 3.3. As *Netto*'s eBPF layer is now notified of every entry and exit from any of these *sub-events*, it can export to the user-space controller a complete record of their execution time, similarly to how it already measures the CPU utilization of other, higher level events.

However, this concept fails to take into account two crucial aspects of the `NET_RX_SOFTIRQ`'s internals:

1. **Execution frequency of its sub-events**: As already pointed out in an earlier paragraph, the `NET_RX_SOFTIRQ` has an upper bound of 300 packets that can be processed per invocation. This limit ensures that the softirq will release the CPU in a timely manner and avoids starvation of other system tasks. At the same time, its packet batching capability is beneficial for *Netto* as the execution frequency of the associated eBPF tracing probes is scaled down with respect to the inbound packet frequency by at most a factor of 300. The reduced eBPF program calls help mitigate their intrinsic system overhead.

   Conversely, most of the `NET_RX_SOFTIRQ`'s sub-events are run for each incoming packet. This translates to possibly millions of times a second for multigig-capable hardware, and especially for UDP or similar connectionless protocols, where GRO/GSO can not artificially aggregate related packets.

   Attaching tens of eBPF probes to such high-frequency hot paths would undeniably magnify *Netto*'s kernel side overhead. The extent of this inefficiency is measured and discussed in Chapter 6.

2. **Complex hierarchy of its sub-events**: The `NET_RX_SOFTIRQ`'s sub-events form a complex hierarchy with their call stacks. This is especially evident for the `netif_receive_skb()` function, which is the core *skb* handling routine and acts as the entry-point to the generic in-kernel networking stack in Linux. This function, through the `br_handle_frame()` receive handler, can in fact be run recursively, in case a bridge interface receives a packet with a destination MAC address matching that of the virtual bridge device.

Because of this and other similar anomalies, the simple execution time measurement of the sub-events by difference of eBPF program invocation timestamps is not adequate anymore. The implemented solution revolves around an eBPF-side *event-stack*; a more detailed description of this algorithm is presented in Chapter 5.

4.3.2 Network Stack Sampling

Algorithm 1 Stack trace symbol counting.

```
1: resetCounts()
2:
3: // Iterate over all captured stack traces
4: for trace in capturedTraces do
5:   // Iterate over all of its instruction pointers
6:   for instructionPointer in trace do
7:     // Compare ip to all known kernel symbol ranges
8:     for sym in knownSymbols do
9:       if instructionPointer in sym.addrRange then
10:        // In case of match, increment associated counter
11:        incrementCountFor(sym)
12:      end if
13:    end for
14:  end for
15: end for
```

A different technique aims to mitigate both of the highlighted complications with Full Functions Tracking, by trading measurement accuracy for lower overhead. Network Stack Sampling works by estimating the cost of the `NET_RX_SOFTIRQ`'s subcomponents instead of attempting an exact measurement; conversely, the eBPF layer remains greatly simplified, as the burden of managing the sub-event hierarchy's complexity is moved to the user-space domain.

The core idea behind Network Stack Sampling is to apply the methodology of sample-based CPU profilers to the Linux kernel's network stack, in real time. These tools, commonly used to debug and diagnose software performance for user-space applications, sample the program's call stack at regular intervals, and then use the captured data to discover hot-paths in the code and suggest optimization efforts.

Netto implements this through a single additional eBPF program of type `perf_event`, which is able to interrupt each CPU core at a user-specified frequency. This parameter is an fundamental configuration knob, as it allows a tailored compromise between measurement accuracy and system overhead. Each invocation of the `perf_event` eBPF program will dump the interrupted core's kernel-side call stack, and pipe

it to the user-space controller for analysis. The acquisition of the stack traces is performed through the dedicated helper functions, and it's thus fairly lightweight. Conversely, the transfer of possibly thousands of stack traces every second by all CPUs — and related retrieval by the controller application — is potentially a large performance bottleneck of this design, and great attention has been put into ensuring that this is implemented as efficiently as possible. In Chapter 5 several viable options are explored and compared.

In any case, the controller will retrieve all captured stack traces during its update cycles. At this point, trace analysis is performed. The strategy is fairly straightforward (see Algorithm 1): for all stack traces, the controller iterates over all the contained instruction pointers, which are then matched up against a predetermined set of kernel symbols, corresponding to the `NET_RX_SOFTIRQ`'s sub-event functions. Every match results in an associated counter being incremented. After completion, the counters are directly used to derive the CPU utilization by relating them to the total execution time of the parent `NET_RX_SOFTIRQ` event for the same time slot.

4.4 Estimating the Total Networking Power Draw

A key application of the network stack diagnostics provided by *Netto* — which is able to measure the CPU utilization associated to networking tasks — is deriving the fraction of the CPU's electrical power consumption due to networking. This feature can help system administrators better understand the energy footprint of their servers, as well as evaluating whether a SmartNIC deployment would prove suitable to reduce the power draw in data centers, as this topic is becoming more and more relevant throughout the years. Indeed, the relationship between networking load and power draw, including the association between possible offloading of the network stack to dedicated accelerators and resulting energy saving, is object of further research, and a possible future direction for the development of *Netto*.

In its current implementation, *Netto* is able to compute a coarse approximation of the network stack's CPU-related power draw by intersecting the whole system instantaneous energy consumption — as reported by Intel RAPL (which was discussed in Section 3.2) — with the fraction of the CPU utilization attributable to the four networking events highlighted earlier in this chapter. In particular, the total electrical power is obtained by integrating the contributions of the *Package* and *DRAM* RAPL domains for each of the available sockets; this figure is then scaled by the overall networking CPU load generated by *Netto* itself and converted to W for presentation as an exported metric.

Indeed, this crude approximation, which is based on an alleged linear relationship between CPU utilization and power output, only roughly resembles actual behavior of modern hardware, that is instead coordinated by many different characteristics such as frequency scaling and C-states. Unfortunately, no higher-resolution power probing interface currently exists to, for instance, return the energy utilization on a

per-core basis, which would allow *Netto* to enhance its energy-related capabilities. In any case, this methodology is also utilized by other energy diagnostic software such as Kepler [7], that also relies on strategically placed eBPF probes to obtain a power scaling factor.

Chapter 5

Netto's Implementation

As of the latest version, *Netto* contains approximately 2000 lines of Rust code for the user-space side of the tool and web frontend (where Rust is compiled into WebAssembly [10] bytecode for execution in a web browser context), coupled with about 300 lines of BPF-C code for the various eBPF probes that power *Netto*'s diagnostic capabilities.

This chapter will expand on the design choices introduced in Chapter 4, showing their implementation details and integrating them with code listings for the various portions of the tool, by discussing the base network event monitoring feature first, then by examining the event breakdown functionality.

Netto's latest source code can be accessed publicly at this GitHub repository: `https://github.com/miolad/netto`. The code is structured as a single Cargo workspace[1], composed of the following members:

- **`netto`**: Main binary crate for the tracing tool. It also contains the BPF-C code in the `netto/src/bpf` folder.
- **`web-frontend`**: WebAssembly binary crate for the custom web frontend that presents the captured metrics in real time to the user.
- **`metrics-common`**: Library crate used to share definitions between `netto` and `web-frontend`.
- **`xtask`**: Meta binary crate used as a development tool to compile and deploy *Netto*. With the alias specified in `.cargo/config.toml`, a pseudo-subcommand

[1]Cargo is the Rust package manager. It is commonly used to aid development of Rust programs thanks to its capability to automatically gather dependencies and configure them through compilation switches (the "*cargo features*"). Workspaces are a special feature of Cargo that allows to group several "*members*" together, each member being a separate *crate*, or project.

is defined as `cargo xtask`, which builds and runs the `xtask` crate with whatever arguments it was given, allowing it to manage the compilation of the other crates.

5.1 eBPF Data Plane

As mentioned above, *Netto*'s eBPF source is located in `netto/src/bpf`, where the tracing code is mainly split into two source files: `prog.bpf.c` and `prog.bpf.h`. `vmlinux.h` is the automatically generated header that contains all the definitions from the Linux kernel for usage by the eBPF code.

The subdivision into header and source files is beneficial as the `.h` contains definitions that must be propagated to the user-space code for the consumption of data contained in eBPF maps. For this reason, `prog.bpf.h` is also read at *Netto* compilation time to generate the equivalent Rust definitions by the popular `bindgen` library crate.

The integration of eBPF code into *Netto*'s user-space program is done through the Rust bindings of the *libbpf* helper library, which implies a CO-RE usage model: this means that at compile time, and in a totally transparent way, Cargo will invoke the *Clang* compiler to build the eBPF code into a binary object file, which is then bundled into the main executable through a *skeleton*, i.e. an automatically generated Rust source file that contains the compiled BPF bytecode as a constant byte array. The skeleton model thus allows accessing eBPF functionality from user code with helpful abstractions over the raw `bpf` system call interface. All the described compile-time behavior is configured in the custom builder script at `netto/build.rs`.

Listing 5.1 shows the entire contents of the `prog.bpf.h` header file. Two notable definitions are:

- `enum event_types`: Enumeration that associates a numeric identifier to each tracked network event.

  A notable difference with respect to the identified events described in Chapter 4 is the presence of a fifth one: `EVENT_IO_WORKER`. This entry refers to time spent by the CPU in `io_uring` worker threads, which do not necessarily run networking tasks, but tracking them can help contextualize the overall CPU utilization of the host computer, especially when `io_uring` has been configured with `IORING_SETUP_SQPOLL`, which dedicates a worker thread to busy polling of the Submission Queue, burning a CPU core.

- `struct per_cpu_data`: Crucial struct that defines the format of captured metrics and how they are transported from the in-kernel eBPF layer to *Netto*'s Rust controller.

```c
#ifndef _PROG_BPF_H_
#define _PROG_BPF_H_

#include "vmlinux.h"

enum event_types {
    EVENT_SOCK_SENDMSG   = 0,
    EVENT_SOCK_RECVMSG   = 1,
    EVENT_NET_TX_SOFTIRQ = 2,
    EVENT_NET_RX_SOFTIRQ = 3,
    EVENT_IO_WORKER      = 4,

    EVENT_MAX            = 5
};

struct per_cpu_data {
    /// @brief Latest entry timestamp to any event in ns
    u64 entry_ts;

    /// @brief Latest scheduler switch timestamp
    u64 sched_switch_ts;

    /// @brief Total CPU time accounted to various events since the last
    scheduler switch
    u64 sched_switch_accounted_time;

    /// @brief Total time in ns registered for each event
    u64 per_event_total_time[EVENT_MAX];

    /// @brief When non-zero, stack traces by the perf event prog are enabled
    u8 enable_stack_trace;
};

#endif
```

Listing 5.1: Entire contents of the `prog.bpf.h` header.

The core of the struct is the pair of fields **`entry_ts`** and **`per_event_total_time`**; the former contains the most recent entry timestamp in ns to any event, and is only used internally by the eBPF probes, while the latter is an array that contains the total cumulative time spent by the CPU in each of the tracked events since *Netto* was loaded into the systems.

The other fields are used to enable the monitoring of the `EVENT_IO_WORKER` (`sched_switch_ts` and `sched_switch_accounted_time`) — as it can be done by exploiting the already existing `sched_switch` tracepoint and no additional probe —, or as part of the final implementation of the `NET_RX_SOFTIRQ` cost

breakdown feature (`enable_stack_trace`), which is going to be discussed later in this chapter.

As the name implies, this struct is instantiated once for every CPU core, in order to support per-CPU metric collection.

5.1.1 Maps Definition

Two eBPF maps support the basic event measurement feature, with a third used for the `NET_RX_SOFTIRQ` breakdown logic, which will be examined in 5.3. The BPF-C code where these maps are defined[2] is reported in Listing 5.2.

```
15 /**
16  * Keeps track of which tasks are currently being tracked
17  * by associating an event identifier to each of them.
18  */
19 struct {
20     __uint(type, BPF_MAP_TYPE_TASK_STORAGE);
21     __type(key, u32);
22     __type(value, u64);
23     __uint(map_flags, BPF_F_NO_PREALLOC);
24 } traced_pids SEC(".maps");
25
26 /**
27  * Per-cpu timestamps and counters
28  */
29 struct {
30     __uint(type, BPF_MAP_TYPE_PERCPU_ARRAY);
31     __uint(key_size, sizeof(u32));
32     __uint(value_size, sizeof(struct per_cpu_data));
33     __uint(max_entries, 1);
34 } per_cpu SEC(".maps");
```

Listing 5.2: `netto/src/bpf/prog.bpf.c`: Definition of the main eBPF maps.

The first one, named `traced_pids` in the code, is a `BPF_MAP_TYPE_TASK_STORAGE` map used to implement the race condition mitigation strategy described in Section 4.2: it stores the event identifier of the currently running networking event on each task for usage by the two softirq probes and the `sched_switch` tracepoint program. The requested map flag `BPF_F_NO_PREALLOC` requires that all reads from a task's eBPF storage fail until an explicit write has been performed by a call to the helper `bpf_task_storage_get()`. This helps ensure correct concurrent behavior by disallowing access to such storage before the first event entry. The semantics of

[2] eBPF maps can be declared directly from eBPF code by allocating specific structures in the "`.maps`" ELF section of the compiled program, describing the requested map. Alternatively, map allocation can be performed from the user-space by interacting with the `bpf` system call.

the task storage contents dedicates the value `EVENT_MAX` to encoding a "no events" case.

The `per_cpu` map instead is a one slot per-CPU array, providing differentiated memory for the multiple CPU cores of the system, where the previously described `per_cpu_data` struct is stored.

5.1.2 Basic Entry-Exit Program Logic

At the heart of *Netto*'s tracing features are the main eBPF programs attached to the entry and exit addresses of the Linux networking entry-points, used to measure their execution time by taking the difference in the probes' invocation timestamps. In total, six such programs are defined in `prog.bpf.c` (as the two softirq networking events share the same couple of tracepoints, as shown in Table 4.1). However, since the overall functionality is mostly the same between the different events, only one specific instance is reported in Listings 5.3 and 5.4, which show the programs attached to the `sock_recvmsg()` function.

```
127 SEC("fentry/sock_recvmsg")
128 int BPF_PROG(sock_recvmsg_entry) {
129     u32 zero = 0;
130     struct per_cpu_data* per_cpu_data;
131     u64* per_task_events, now = bpf_ktime_get_ns();
132 
133     if (
134         likely((per_task_events = bpf_task_storage_get(&traced_pids,
    bpf_get_current_task_btf(), &event_max, BPF_LOCAL_STORAGE_GET_F_CREATE))
    != NULL) &&
135         likely((per_cpu_data = bpf_map_lookup_elem(&per_cpu, &zero)) != NULL)
136     ) {
137         per_cpu_data->entry_ts = now;
138         *per_task_events = EVENT_SOCK_RECVMSG;
139     }
140 
141     return 0;
142 }
```

Listing 5.3: `netto/src/bpf/prog.bpf.c`: Entry eBPF program for the socket receive network event.

Entry programs are very simple: after acquiring a reference to both maps' memory, the probe updates the CPU's latest entry timestamp and then sets the current event as active for the underlying task.

```
144 SEC("fexit/sock_recvmsg")
145 int BPF_PROG(sock_recvmsg_exit) {
146     u32 zero = 0;
147     struct per_cpu_data* per_cpu_data;
148     u64* per_task_events, now, entry_ts, t;
```

```
149
150     if (
151         likely((per_task_events = bpf_task_storage_get(&traced_pids,
        bpf_get_current_task_btf(), NULL, 0)) != NULL) &&
152         likely((per_cpu_data = bpf_map_lookup_elem(&per_cpu, &zero)) != NULL)
153     ) {
154         entry_ts = per_cpu_data->entry_ts;
155         now = bpf_ktime_get_ns();
156         t = now - entry_ts;
157
158         *per_task_events = EVENT_MAX;
159         per_cpu_data->per_event_total_time[EVENT_SOCK_RECVMSG] += t;
160         per_cpu_data->sched_switch_accounted_time += t;
161     }
162
163     return 0;
164 }
```

Listing 5.4: `netto/src/bpf/prog.bpf.c`: Exit eBPF program for the socket receive network event.

Conversely, exit programs perform the opposite operations to reset the event identifier on the running task and increment its total execution time in the `per_cpu` struct, which is going to be reflected on the final CPU utilization metrics for the associated event. Notice how the event's on-CPU time `t` is computed by subtracting the latest entry timestamp from the current time (returned by the `bpf_ktime_get_ns()` helper). The specific way in which `t` is calculated, by first moving into the local stack the `entry_ts` variable and then `now`, mitigates a rare race condition which happened whenever the `fexit/sock_recvmsg` or `fexit/sock_sendmsg` programs were interrupted by a networking softirq (and thus the attached pair of eBPF programs as well). In such case, it was possible for the `entry_ts` register to be overwritten by the `tp_btf/softirq_exit` eBPF program (as part of its synchronization handling algorithm described in Section 4.2.1) *before* the value could be read by the interrupted probe but *after* the invocation timestamp was obtained from `bpf_ktime_get_ns()`, inducing a negative `t` that would manifest as an instantaneous spike in the measured CPU utilization. This occurrence, albeit rare (but still relatively frequent on high core count systems) is thus resolved by lines 154 to 156 of the above code snippet.

5.1.3 The `sched_switch` Tracepoint

Finally, to conclude the discussion about the in-kernel eBPF component of *Netto*, a mention of the `sched_switch` tracepoint program is in order. As specified in the Architecture chapter of this thesis, such program was made necessary by the need to track migratable tasks across core boundaries. Listing 5.5 displays its code.

```
217 SEC("tp_btf/sched_switch")
218 int BPF_PROG(tp_sched_switch, bool preempt, struct task_struct* prev, struct
        task_struct* next) {
219     u32 zero = 0;
220     struct per_cpu_data* per_cpu_data;
221     u64* prev_task_events, * next_task_events, now = bpf_ktime_get_ns();
222
223     prev_task_events = bpf_task_storage_get(&traced_pids, prev, NULL, 0);
224     next_task_events = bpf_task_storage_get(&traced_pids, next, NULL, 0);
225     per_cpu_data     = bpf_map_lookup_elem(&per_cpu, &zero);
226
227     if (likely(per_cpu_data != NULL)) {
228         if (prev_task_events != NULL)
229             stop_event(*prev_task_events, per_cpu_data, now);
230         if (next_task_events != NULL && *next_task_events != EVENT_MAX)
231             per_cpu_data->entry_ts = now;
232
233         if (prev->flags & 0x10 /* PF_IO_WORKER */)
234             per_cpu_data->per_event_total_time[EVENT_IO_WORKER] += now -
235                 per_cpu_data->sched_switch_ts                           -
236                 per_cpu_data->sched_switch_accounted_time;
237         per_cpu_data->sched_switch_ts = now;
238         per_cpu_data->sched_switch_accounted_time = 0;
239     }
240
241     return 0;
242 }
```

Listing 5.5: `netto/src/bpf/prog.bpf.c`: eBPF probe attached to the `sched_switch` tracepoint.

This probe is also quite straightforward:

1. First the event identifiers of the exiting (`prev`) and entering (`next`) tasks are acquired, together with a reference to this CPU's `struct per_cpu_data` instance (lines 223 to 225).

2. Then, the event ids are used to respectively stop their accounting (line 229: the `stop_event` inline function is a convenience routine to increment an event's total time based on the current invocation timestamp, as seen in the `fexit/sock_recvmsg` program reported above) and update the latest entry timestamp (line 231), if necessary.

3. Finally, the `EVENT_IO_WORKER` pseudo-event is addressed by checking whether the `prev` task was indeed an `io_uring` worker thread, and thus incrementing its total execution time (lines 233 and 234).

5.2 User-Space Control Plane

On the user-space side, the implementation is based on the *actor* abstraction offered by the well known *Actix* crate (https://actix.rs/), a high performance library for building fast concurrent applications in Rust. The actor model works by splitting the business logic of the program into multiple isolated entities (the *actors*) that each implement their own function, thus achieving a separation of concerns that is beneficial for maintainability and overall code quality.

Each actor is identified by a unique *address*, that can be used as the endpoint to which to send *messages*. Message passing is indeed the primary way with which independent actors communicate by exchanging data and delivering signals; each actor can in fact be configured to support the reception of messages based on their static type. The reception of a supported message will then trigger then execution of a handler function in the actor's context, possibly returning a result to the original sender.

The whole actor scaffold is supported by an asynchronous runtime powered by the *Tokio* crate (https://tokio.rs/), a state-of-the-art implementation of the async execution model for the Rust programming language, based on a multi-threaded work-stealing scheduler that allows to efficiently host multiple tasks on the available kernel threads. In *Netto*'s case, the Tokio engine is kept single-threaded for simplicity, and because its specific workload would not benefit from parallel execution.

All of *Netto*'s actors are found in the `netto/src/actors` folder, whereas the defined message types used for inter-actor communications are in `netto/src/actors/mod.rs`. The following subsections will describe the implemented actors and how they interact with each other.

5.2.1 `TraceAnalyzer`

This is the primary entity responsible for interacting with the eBPF core and exporting metrics. After the startup initialization — where eBPF programs are loaded and attached to their respective hook points in the kernel — ownership of the libbpf skeleton is transferred to this actor, which is hence capable of accessing the `per_cpu` map where eBPF metrics are stored.

The core of this module is `TraceAnalyzer`'s `run_interval()` method, which is periodically run by Actix based on the controller update cycle frequency parameter (which defaults to 500 ms). Each invocation of the `run_interval()` method will:

1. Compute variations in time and total electrical energy with respect to the previous execution of the function.

   The former should, under normal working conditions, closely resemble the requested period provided by the user configuration, but it could sway slightly

due to scheduling inaccuracies, especially in highly loaded systems. In order to provide the maximum accuracy when computing the final metrics, this value is thus measured.

The latter allows *Netto* to derive the average power drawn by the system since the previous controller iteration, which is used to implement the overall networking power estimation, as explained in Section 4.4.

```
let delta_time = {
    let dt = now.duration_since(self.prev_update_ts);
    self.prev_update_ts = now;
    dt
};
let delta_energy = self.rapl.as_ref().map(|rapl| {
    let current_total_energy = rapl
        .sockets
        .values()
        .flat_map(|socket| socket.energy())
        .sum();
    let delta_energy = current_total_energy - self.prev_total_energy;
    self.prev_total_energy = current_total_energy;
    delta_energy
});
```

Listing 5.6: `netto/src/actors/trace_analyzer.rs`: Computation of the delta time and energy.

2. Handle the stream of captured stack traces from eBPF to implement *Network Stack Sampling* (discussed later). This step names the entire actor.

3. Perform the lookup into the `per_cpu` eBPF map and integrate the raw values with those from the previous controller iteration to produce the CPU utilization metrics.

   In this step the controller iterates over the `per_event_total_time` array; for each entry, it chooses a descriptive name and emits the metric by sending a `MetricUpdate` message to the `MetricsCollector` actor.

   Finally, the `MetricsCollector` is notified about the event array being consumed by a `SubmitUpdate` message, which also includes the updated networking power draw and the generic `/proc/stat` metrics, captured and exported for validation concerns.

```
self.metrics_collector_addr.do_send(SubmitUpdate {
    net_power_w: delta_energy.map(|e| (e as f64) *
        total_cpu_frac / (
            delta_time.as_secs_f64() * 1_000_000.0
        )
    ),
```

```
352     user_space_overhead: now.elapsed().as_secs_f64() /
353         delta_time.as_secs_f64(),
354     procfs_metrics
355 });
```

Listing 5.7: `netto/src/actors/trace_analyzer.rs`: Final notification to the `MetricsCollector` actor, signaling a successful control iteration.

5.2.2 `MetricsCollector`

The `MetricsCollector`'s purpose is twofold: (*i*) it serves as the repository to store CPU consumption metrics as they are computed by the `TraceAnalyzer` actor, and (*ii*) it also pushes the completed updates to any configured client.

To do these tasks, the actor implements handlers for two main message types, `MetricUpdate` and `SubmitUpdate`, both of which have already been mentioned in the previous subsection. The former carries the latest information about one specific metric, such as its **name**, average **CPU time fraction** for the latest controller update cycle, and **CPU id** of the processor core to which the data refers to. Upon receiving such a message, the `MetricsCollector` stores the updated information about the metric in an inner and cohesive representation of the whole metrics tree, using the name field of the message as a persistent and hierarchical path that identifies the metric across controller iteration boundaries (the hierarchical nature of the name property is used for nested metrics, common due to the `NET_RX_SOFTIRQ` breakdown functionality. For instance, the name "*RX Softirq/Bridging*" identifies the *Bridging* submetric of the *RX Softirq* event). Notably, the `MetricsCollector` actor's code is completely agnostic to the actual metrics produced by the system, as it will seamlessly support metrics that dynamically change at runtime, although this use case is not currently exploited by the rest of the application.

As the `TraceAnalyzer` scans the `per_cpu` eBPF map, metrics are produced and pushed to the `MetricsCollector` for storage. Eventually, when the map has been drained, a single `SubmitUpdate` message is sent to the `MetricsCollector`; this message signals the completion of the controller iteration, triggering the propagation of the metrics tree to all clients. Currently *Netto* supports three client types: a file-based logger, a Prometheus-compatible exporter, and a custom web solution designed for real-time consumption of the data. The remaining actors implement these clients.

5.2.3 `WebsocketClient`

Netto's custom web frontend works by serving a static HTML document with accompanying WebAssembly script (the `web-frontend` binary crate) via Actix's built-in HTTP server. Upon loading the page on the client's browser, the wasm program

loads and establishes a WebSocket connection to *Netto*, through which all metrics updates are streamed. More details on this solution are provided in 5.4.

The `WebsocketClient` actor represents a single WebSocket connection to a web client; upon upgrade of the HTTP socket, Actix automatically instantiates a corresponding `WebsocketClient`, that will then register itself to the singleton `MetricsCollector`. As the `SubmitUpdate` messages are relayed to all registered `WebsocketClients`, a *Message Pack*[3] encoded version is sent over the connection to the wasm script running in the client browser.

5.2.4 `FileLogger`

The `FileLogger` implements a basic log-to-file functionality for *Netto*, where all metrics are dumped to a user-specified file. Similarly to the previous `WebsocketClient`, the Message Pack encoding format is also used, not only because of the low resource utilization, but mostly due to the constant size of each metrics snapshot, which greatly simplifies decoding of the log. The typical bandwidth of a *Netto* log is around 3.3 kbps per core.

5.2.5 `PrometheusLogger`

Finally, the `PrometheusLogger` concludes the list of used actors. As the name implies, it organizes available metrics into a Prometheus registry and encodes them to text to be served by the same HTTP server that manages the custom frontend. Prometheus (https://prometheus.io/) is a successful solution for managing custom time series metrics, which also defines an efficient format that has become a de-facto standard throughout the years. Compatibility with Prometheus provides *Netto* with the ability to interface with powerful data visualization and management software such as the Grafana (https://grafana.com/) product stack.

5.3 Event Breakdown

Netto is capable of producing a breakdown of the cost of the `NET_RX_SOFTIRQ` network event into its main components. This is particularly useful because the receive-side networking stack embedded in this softirq hides several core network functions that could potentially be stressed more or less depending on the specific load applied to the host system. Knowing what component is causing a large CPU utilization hit may allow system administrators to more accurately optimize their

[3]Message Pack is an efficient binary encoding format that was chosen for *Netto* due to its low overhead both in message size and computational power required for serialization and deserialization. https://msgpack.org/

servers. The greater diagnostic resolution provided by the event breakdown feature is thus invaluable for a large variety of use cases, and in the future *Netto* could be expanded to support more granular breakdowns of other events beyond the `NET_RX_SOFTIRQ`.

Currently, *Netto* supports the following `NET_RX_SOFTIRQ` sub-events:

1. **Driver Poll**: It represents the overhead associated with polling the NIC driver.

   Since all drivers use a different polling function, this metric is quite opaque, as the amount and type of work performed might differ depending on the host hardware. Typically, though, it involves retrieving packet data from dedicated memory buffers, allocating or recycling skbs to store them, and several other operations. During testing of *Netto*, this metric would often stand out as one of the most prevalent contributions to the overall softirq's cost.

2. **GRO Overhead**: *Generic Receive Offload* (GRO) is a hardware-backed technology that allows reconstruction of network flows for stream-oriented transport protocols such as TCP by the NIC. GRO is crucial to enhance the efficiency of the TCP/IP receive-side stack specifically as it can condense multiple segments of the same network flow into bigger skbs, thus reducing the per-packet computational overhead. Drivers of GRO-capable NICs use special functions to submit skbs to the common upper layers of the network stack such as `napi_gro_receive()`, instead of the more generic `netif_receive_skb()`. These variants incur into some level of CPU overhead due to the packet management tasks they provide, which this metric attempts to isolate. In practice, although sometimes significant, any amount of overhead induced by GRO is less than the cost of disabling the feature.

3. **XDP Generic**: This sub-event measures the cumulative CPU consumption of any eBPF program attached to the `XDP_GENERIC` attachment point. As opposed to the `NATIVE` or `OFFLOADED` counterparts, `XDP_GENERIC` is hosted in the common networking path, thus allowing *Netto* to target it.

4. **TC Classify**: It refers to the classification step of the *Traffic Control* Linux subsystem.

5. **NF Ingress**: This is NetFilter's earliest hook point in the receive-side of the network stack, and as such it can be used to implement early packet filtering.

6. **Conntrack**: Conntrack is an important element of the NetFilter firewall architecture, which is in charge of connection tracking (as the name suggests). This makes NetFilter a stateful firewall. Conntrack's cost can be significant in some specific circumstances.

7. **Bridging**: A software bridge is a networking device that implements the switching algorithm for L2 traffic; in Linux, a bridge is defined by means

of which interfaces are part of it. The CPU cost of a software bridge can be attributed to parsing of the Ethernet headers and forwarding frames to output interfaces.

8. **NF Prerouting**: Another component of the NetFilter architecture, the `PREROUTING` stage refers to the instant before any routing decision takes place, in this case for traffic in the input chain.

   Since this hook is specific for IP packets, two submetrics are actually defined, for IPv4 and IPv6 respectively.

9. **Forwarding**: Network function applied to inbound IP packets whose destination L3 address is a that of a remote host. Forwarding is typically globally disabled on end hosts through a specific configuration switch, but in modern servers this is likely the most prominent networking function used, as large amounts of data can be exchanged by different VMs or network namespaces.

10. **Local Delivery**: Again referring to inbound IP traffic, the term "*local delivery*" is used to describe packets directed to local applications. This sub-event (which is also split in a v4 and v6 version) thus includes all CPU cycles spent bringing the received packets to their destination socket, hence including possible processing of higher level protocols.

In Section 4.3 two different mechanisms were discussed as possible implementations for this feature. Now, Full Functions Tracking and Network Stack Sampling are more accurately examined and presented together with their implementation details.

5.3.1 Full Functions Tracking

As already pointed out in a previous chapter, Full Functions Tracking suffers from two crucial flaws: (*i*) poor performance due to amplified eBPF instrumentation overhead for kernel's hot functions, and (*ii*) inadequate scalability caused by having to deal with the sub-events hierarchy complications directly in eBPF. And while it is certainly the first of the two issues (whose extent is going to be analyzed in Chapter 6) that would eventually lead to the abandonment of this option in favour of the more elegant Network Stack Sampling solution, the second problem is undoubtedly an interesting engineering hurdle that deserves some further discussion.

Indeed, with the ambition of designing a fully generic algorithm that would thus be applicable to any set of sub-events (a crucial requirement to maintain scalability and compatibility with past and future versions of the Linux kernel), it is of essential importance to consider edge cases where the structure of the targeted sub-events (i.e. what is the relationship of their functions and how they are linked together) is not a simple tree, but rather a more generic graph.

This poses critical problems to how the CPU time is accounted for the various networking events, as the semantics of whether a new event entry should replace or stack on top of a pending event is a case-by-case decision that depends on the specific events considered. Additionally, some events could also be run recursively, aggravating the situation further. This behavior is triggered, for example, by the implementation of the bridging functionality, which can occasionally "*pass frames up*", causing the recursive execution of the `netif_receive_skb()` function.

In *Netto*'s implementation of Full Functions Tracking (which can be consulted at the aptly named branch of the main GitHub repository: https://github.com/miolad/netto/tree/full-functions-tracking), these problems were addressed by a dedicated "*event stack*": a per-task structure that would store the ordered list of the currently running networking events (Listing 5.8).

```
47 /**
48  * Contains a stack of the currently in-flight events
49  * for a given task
50  */
51 struct event_stack {
52     /// @brief Each element is an index into the events array
53     u16 stack[EVENT_STACK_SIZE];
54     /// @brief Index of the first empty frame in the stack
55     u16 stack_ptr;
56 };
```

Listing 5.8: `main/src/bpf/event_stack.bpf.h`: Definition of the `event_stack` structure, a key component of the Full Functions Tracking architecture.

Furthermore, access to the event stack should be synchronized between concurrent invocations of eBPF programs, not because of multithreading concerns (as the event stack is a per-task property, which is thus only ever hosted on one CPU core at a time), but due to IRQ-hosted events such as `NET_RX_SOFTIRQ` and `NET_TX_SOFTIRQ` possibly interrupting other eBPF programs. This is achieved in the code by means of a `bpf_spin_lock` object.

Two operations are supported on the event stack structure, through which the eBPF probes would update the respective events' measured CPU time:

- **Push**: Pushes a new event to the top of the stack, possibly updating the timings of any other event in the stack. Note that this requires scanning the stack, and is thus a linear operation with respect to the stack size. In the code (not reported here due to space constraints), some tricks were required to ensure correctness, such as explicitly checking for specific events or using often convoluted expressions to persuade the eBPF verifier (usually a symptom of high code complexity).
- **Pop**: The opposite operation of the previous push, it removes the topmost event from the stack. Similarly to the push operation, equivalent issues plague

the pop as well.

On the user-space controller side, the implications of Full Functions Tracking-based event breakdown are minimal, as the final metrics are exported directly by the eBPF layer, no differently from how the four main networking events were handled previously.

5.3.2 Network Stack Sampling

The Network Stack Sampling strategy is a more clever solution for the event breakdown functionality that sidesteps most of the complications of Full Functions Tracking. Specifically, the large complexity associated with sub-event hierarchy — which required the utilization of a cumbersome event stack before — is now lifted from the eBPF layer and into user code, where *Netto* does not have to deal with eBPF restrictions such as limited stack size or verifier constraints. Additionally, removing a major source of complexity from the in-kernel critical data path, along with getting rid of the expensive eBPF instrumentation for most of the hot functions, is going to inevitably determine a substantial performance advantage with respect to the previous attempt.

Of course, as explained in the dedicated section of Chapter 4, the drawback of Network Stack Sampling is manifested as a theoretically reduced accuracy of the sub-event metrics, by a factor that is directly related to the configured sampling frequency setting. Conversely, Network Stack Sampling is however an unbiased algorithm, meaning that no inherent error is introduced in the estimation, and increasing the aforementioned frequency would further converge to the exact solution. In practice, though, this apparent disadvantage when compared with Full Functions Tracking is overshadowed by the absolute cost of the previous technique, which would also skew the measurements (this is covered in detail in the next chapter).

Network Stack Sampling only requires a single additional eBPF program on top of the ones introduced for the basic measurement feature of the four base networking events. This is the `perf_event` probe that — through periodic invocation on all the CPU cores available in the system — is responsible for dumping the valuable kernel-side stack traces for later analysis by the user-space controller. Listing 5.9 reports the code of this extra program.

```
240 SEC("perf_event")
241 int perf_event_prog(struct bpf_perf_event_data* ctx) {
242     struct per_cpu_data* per_cpu_data;
243     u32 index, zero = 0;
244     u64* buf;
245 
246     if (
247         likely((per_cpu_data = bpf_map_lookup_elem(&per_cpu,
248             &zero)) != NULL) &&
```

```
249         per_cpu_data->enable_stack_trace
250     ) {
251         index = __sync_fetch_and_add(
252             stack_traces_slot_off ? &stack_traces_count_slot_1 :
253                                     &stack_traces_count_slot_0,
254             1
255         ) + stack_traces_slot_off;
256 
257         if (likely((
258             buf = bpf_map_lookup_elem(&stack_traces, &index)
259         ) != NULL)) {
260             *buf = (u64)bpf_get_smp_processor_id() |
261                 ((u64)bpf_get_stack(
262                     ctx,
263                     buf + 1,
264                     sizeof(u64) * 127,
265                     0) << 32
266                 );
267         }
268     }
269 
270     return 0;
271 }
```

Listing 5.9: `netto/src/bpf/prog.bpf.c`: The `perf_event` eBPF program that periodically interrupts the CPU and captures the kernel-side stack trace.

The snippet, which is relative to the *mmapable array* implementation (refer to 5.3.2 for details), shows a very simple structure. After acquiring a suitable array slot to save the stack trace to, the `bpf_get_stack()` helper is used to perform the actual stack walking and writing the stack frame data to memory, together with critical information such as the processor id and size of the captured trace. The compact nature of the program helps keep it fast and efficient, as the evaluation chapter will highlight.

One notable optimization is the condition on line 249: only during the execution of a `NET_RX_SOFTIRQ` *Netto* is interested in capturing stack traces, and so the feature is automatically enabled on-demand only for CPU cores busy serving the above softirq. This reduces the flow of stack traces that must be analyzed, optimizing the controller's CPU load during idle networking conditions.

Finally, Listing 5.10 shows how the `perf_event` programs are loaded and attached. For each CPU in the system, an identical `perf_event` is opened with `type PERF_TYPE_SOFTWARE` and `config PERF_COUNT_SW_CPU_CLOCK`; this determines a synthetic event that is triggered at the requested rate. Then, the `perf_event_prog` showed above is attached to each of the individual events through the file descriptor returned by the `perf_event_open` system call.

```rust
// Open and attach a perf-event program for each CPU
let _perf_event_links = unsafe {
    let iter = (0..num_possible_cpus)
        .map(|cpuid| {
            let mut attrs = perf_event_attr {
                size: std::mem::size_of::<perf_event_attr>() as _,
                type_: PERF_TYPE_SOFTWARE,
                config: PERF_COUNT_SW_CPU_CLOCK as _,

                // Sampling frequency
                __bindgen_anon_1: perf_event_attr__bindgen_ty_1 {
                    sample_freq: cli.frequency
                },

                ..Default::default()
            };

            // Only count kernel-space events
            attrs.set_exclude_user(1);

            // Use frequency instead of period
            attrs.set_freq(1);

            (cpuid, attrs)
        });

    let mut v = Vec::with_capacity(num_possible_cpus);
    for (cpuid, mut attrs) in iter {
        // Open the perf-event
        let fd = perf_event_open(&mut attrs, -1, cpuid as _, -1, 0);
        if fd < 0 {
            return Err(std::io::Error::last_os_error().into());
        }

        // Attach to BPF prog
        v.push(skel.progs_mut().perf_event_prog().attach_perf_event(fd)?);
    }

    v
};
```

Listing 5.10: `netto/src/main.rs`: Opening of the `perf_event`s and attachment of the associated eBPF program for each CPU.

Stack Trace Map Choice

Listing 5.9 anticipated that the current implementation of *Netto* uses a `BPF_MAP_TYPE_ARRAY` with the accompanying `BPF_F_MMAPABLE` flag to "*lift*" the captured stack traces

from the in-kernel eBPF layer to the user-space controller for analysis, but multiple different choices have been considered in its place during development, some of which are now described here.

- `BPF_MAP_TYPE_STACK_TRACE` + `BPF_MAP_TYPE_HASH`: The *stack trace* map type is a special purpose structure that is meant as specialized storage for stack traces captured from the dedicated eBPF helpers. In particular, the `bpf_get_stackid()` function will perform the stack dump efficiently in native kernel code, as well as hashing the captured data into a "*stackid*", which is then associated to the trace; the captured bytes are automatically stored in the provided stack trace map, where it can be later retrieved from the user-space controller via the `bpf` system call. A second map of type `BPF_MAP_TYPE_HASH` is also required to supplement each stackid with the number of identical traces captured.

  The benefit of this approach lies in the automatic *in-kernel trace summarization* feature provided by the stack trace map, which reduces the amount of unique traces that must be independently analyzed by the application. Conversely, this solution turned out to be severely inadequate for *Netto*'s use-case, as the stack trace map type lacks any efficient method of bulk retrieval of its entries from the user-space: in its current state, such a solution would require several system calls *per stackid* to read and delete from both maps. Considering that several thousands traces can be produced every second (a number that also scales linearly with the amount of available CPU cores), the system call overhead would simply be unacceptable.

- **Single** `BPF_MAP_TYPE_HASH`: Conceptually, a `BPF_MAP_TYPE_STACK_TRACE` is just a hash map that can only store stack traces keyed by their own stackid. It must thus be possible to "emulate" its behavior with a more generic hash map, which could then also alleviate the requirement for a separate map to store trace counts, as the same information could more cleverly be packed into the single value type.

  The real advantage of this method, however, is the support for batch lookups through the `bpf` system call, which enables the application to potentially dump the entire contents of the map to the user address space with a single transition to the kernel and back, dramatically enhancing the efficiency of the trace retrieval operation. Unfortunately, the downside of replacing the dedicated stack trace map is a far greater trace capture overhead in eBPF caused by having to effectively capture each trace twice. This is due to the helper `bpf_get_stackid()` — which represents the only way to obtain a stackid — only supporting `BPF_MAP_TYPE_STACK_TRACE` as its target for trace storage.

  Although undoubtedly better than the previous suggestion, the single hash

map solution is hardly the best option when renouncing *in-kernel trace summarization*.

- `BPF_MAP_TYPE_RINGBUF`: The eBPF ring buffer is perhaps the most idiomatic way to stream potentially large amounts of binary data from eBPF to a user-space consumer, and as such it must be considered for this use case. In *Netto*'s online GitHub repository, an implementation based on this map type is available under the `perf-event-ringbuf-reserve` branch.

  Over its predecessor (the `BPF_MAP_TYPE_PERF_EVENT_ARRAY`), the bpf ring buffer improves performance and usability, and should thus always be preferred. In particular, the new "reserve-commit" API allows the eBPF producer to reserve a slice of the buffer to write data to, often avoiding an extra copy of the message. *Netto* exploits this functionality at the expense of variable sized messages; the `bpf_ringbuf_output()` helper could in fact push messages of variable sizes to the ring, by needing to copy their contents instead.

  Finally, the ring buffer's "*smart wakeup*" feature — which is capable of waking up consumers upon reception of a message — is not useful here as the map will be completely drained during every controller update iteration.

- **mmapable** `BPF_MAP_TYPE_ARRAY`: The last of the discussed solutions (and the one actually implemented in upstream *Netto*), is a lower-level, more tailored version of the above ring buffer. A bpf array declared with the `BPF_F_MMAPABLE` flag essentially acts as a raw memory buffer that is directly accessible both by eBPF *and* user-space code, effectively negating any cost of stack trace retrieval by the controller.

```
48 struct {
49     __uint(type, BPF_MAP_TYPE_ARRAY);
50     __uint(map_flags, BPF_F_MMAPABLE);
51     __uint(key_size, sizeof(u32));
52     __uint(value_size, sizeof(u64)*128);
53     __uint(max_entries, 1);
54 } stack_traces SEC(".maps");
```

Listing 5.11: `netto/src/bpf/prog.bpf.c`: Declaration of the *mmapable* array map to store and lift the captured CPU stack traces for Network Stack Sampling.

In the code listing above, that shows the declaration of the map in eBPF, the `max_entries` field is set to one only as a placeholder, as the actual size of the map is calculated at runtime (based on CPU core count and dynamic configuration settings) and applied before the map is loaded by libbpf. Each entry encodes the size of the trace and CPU id in its first byte, and the actual trace in the remaining 127 B (default maximum stack trace size in Linux). The map is logically split in two slots, implementing a double buffered architecture:

whenever the controller drains the map, it will scan it from the beginning to the address of the last inserted entry. During this operation, however, more traces might be added by eBPF, as the kernel and application domain of *Netto* run asynchronously to each other. In a normal single-buffered scenario, this determines the loss of the newly inserted entries. By utilizing double buffering instead, there will always be one active slot for writing by eBPF and the other for trace consumption by the controller, for them to be swapped at every control loop cycle.

The currently active slot, together with a counter of the filled entries for each slot, is stored in purposefully allocated global variables of the eBPF program (eBPF global variables are shared between all programs in their source file, as well as the controller application).

```
152 // Swap buffer slots and get the number of stack traces
153 // in the previously active slot
154 let slot_off = self.skel.bss().stack_traces_slot_off as usize;
155 let num_traces_ref;
156 (self.skel.bss().stack_traces_slot_off, num_traces_ref) =
157     if slot_off > 0 {
158         (0, &mut self.skel.bss().stack_traces_count_slot_1)
159     } else {
160         (
161             self.stack_traces_slot_size,
162             &mut self.skel.bss().stack_traces_count_slot_0
163         )
164     };
165
166 // Make sure to read the count *after* swapping the slots
167 let num_traces = *num_traces_ref;
168
169 ... Snippet (perform trace analysis) ...
170
171 // Reset the stack traces index for this slot
172 *num_traces_ref = 0;
```

Listing 5.12: `netto/src/actors/trace_analyzer.rs`: How the *mmapable* array is accessed by the controller to drain the accumulated stack traces.

Trace Analysis

During each periodic controller iteration, the stack traces produced by eBPF will be retrieved and "analyzed", which involves looping over all of their contained instruction pointers and matching them against a set of selected kernel symbols. The instruction pointers (or "*frames*") that make up a stack trace represent the return addresses that each function call automatically pushes to the stack, and can thus be used to reconstruct the steps that led to the moment when the trace

was captured (excluding functions that have been compiled *inline*). Each frame can therefore be directly associated to a specific function of the traced software (the kernel itself in this context), as it lies within the addresses of its first and last instructions.

Upon the initial loading of *Netto*, the virtual file `/proc/kallsyms` is scanned to build a balanced tree of the required sub-event functions, which will later be used to map instruction pointers from stack traces to their respective functions. The `/proc/kallsyms` file contains, for each exported symbol of the running kernel and loaded modules, its name and memory address; for functions, this is their starting address. Since *Netto* needs the full range of memory addresses where a function is accessible, the ending address is derived relative to the position of the subsequent symbol in memory. After the first linear scan of the file, *Netto* proceeds with a filtering step, to only retain the small subset of symbols that are needed by the `NET_RX_SOFTIRQ` breakdown functionality, such as `br_handle_frame` or `do_xdp_generic`; failing to prune away useless symbols would needlessly aggravate runtime performance. Finally, the remaining entries are placed in a dedicated balanced tree (which keyes them by their starting address), that allows fast, logarithmic time access for each instruction pointer. In the code, the `KSyms` structure (`netto/src/ksyms.rs`) serves as a wrapper around such tree (Listing 5.13). The `fun` field of `struct KSymsVal` functionally represents a function pointer that describes what counters must be incremented in case of a match.

```
7  pub struct KSyms {
8      syms: BTreeMap<u64, KSymsVal>
9  }
10
11 type SymbolFun = Box<dyn for<'a>
12     Fn(&'a mut Counts, &'a mut PerFrameProps) -> Option<&'a mut u16>
13 >;
14
15 struct KSymsVal {
16     range_end: u64,
17     fun: SymbolFun
18 }
```

Listing 5.13: `netto/src/ksyms.rs`: Definition of the `KSyms` structure.

Finally, at runtime, the core of the trace analysis routine is reported in Listing 5.14.

5.4 Data Presentation

A crucial part of any diagnostic software is how the data it produces is presented to the final user. In *Netto*, three separate output interfaces are available, all of which have already been introduced when discussing their respective Actix actors: file

```
196 for frame_idx in 0..max_frames {
197     // Load stack frame
198     let ip = trace_ptr.add(frame_idx).read_volatile();
199     if ip == 0 {
200         break;
201     }
202
203     // Check for known symbols
204     if let Some((_, KSymsVal { range_end, fun })) = ksyms
205         .syms
206         .range(..=ip)
207         .next_back() {
208             if ip < *range_end {
209                 if let Some(cnt) = fun(&mut c, &mut frame_props) {
210                     *cnt = 1;
211                 }
212             }
213         }
214 }
```

Listing 5.14: `netto/src/ksyms.rs`: Core of the trace analysis routine.

logging, Prometheus-compatible logging, and a custom web frontend. This section expands on the latter option, presenting a more detailed implementation overview, along with discussing its features.

The custom web frontend is designed as an ideal interface for real time consumption of the CPU utilization metrics, and as such it proved particularly helpful during development and validation of the tool, whereas stable deployments of *Netto* will probably settle for the Prometheus exporter. The main visualizations are a table where all events and sub-events are hosted (Figure 5.1), and a bar graph that updates in real time to reflect the system's utilization.

The implementation is based on the static delivery of a bare-bones HTML skeleton (`www/index.html`), which calls for the loading of an external module script, compiled into WebAssembly from the `web-frontend` crate. The script will then connect via WebSocket back to *Netto*, in order to receive the metrics updates as soon as they are produced. The frontend is hence able to deserialize the received messages from the Message Pack format and use the data to update the page's contents by directly manipulating its DOM.

For the generation of the graph (which Chapter 6 has various examples of), *Netto* uses the *Plotters* library (https://plotters-rs.github.io/home/#!/) to render it directly into an SVG container. The plot is composed of three vertical bars that present successive levels of detail into the running system. The leftmost one presents the overall distribution of the CPU between system, user, and idle modes, as reported by the `/proc/stat` metrics; then, the middle bar zooms into

Metrics\CPUs	CPU0	CPU1	CPU2	CPU3	CPU4	CPU5	CPU6	CPU7	Cumulative
TX syscalls	0.00%	0.00%	0.01%	0.00%	0.00%	0.00%	0.00%	0.01%	0.00%
RX syscalls	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
TX softirq	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
RX softirq	0.00%	0.00%	0.01%	0.00%	0.01%	0.01%	0.00%	0.00%	0.00%
├ Driver poll	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
├ GRO overhead	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
├ XDP generic	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
├ TC classify	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
├ NF ingress	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
├ NF conntrack	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
├ Bridging	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
├ NF prerouting									
│ ├ v4	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
│ └ v6	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
├ Forwarding									
│ ├ v4	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
│ └ v6	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
└ Local delivery									
├ v4	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
└ v6	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
IO workers	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
TOTAL	0.00%	0.00%	0.02%	0.00%	0.01%	0.01%	0.00%	0.01%	0.01%

Figure 5.1: Snapshot of the overview table of *Netto*'s custom frontend. The screenshot was captured on an idle 8-core system.

the kernel section, relating *Netto*'s base networking events to the total amount of CPU time spent in the kernel. Finally, the remaining bar shows the `NET_RX_SOFTIRQ` breakdown into its main components.

Chapter 6

Results and Validation

To ensure the quality and correctness of the captured metrics, *Netto* has been subjected to testing and validation against a set of controlled workloads, where a reasonable expectation of the measurement outcome could be formed prior to running the tests. Additionally, the `/proc/stat` values have been used as ground truth on which to base the interpretation of *Netto*'s output: even though `/proc/stat` only provides generic system diagnosis capabilities — and is not in any way specific to the Linux networking subsystem — there should still be some correlation between its and *Netto*'s results.

First of all, however, Section 6.1 reports a formal comparison between Full Functions Tracking and Network Stack Sampling, the two different proposed methods for implementing the `NET_RX_SOFTIRQ` cost breakdown functionality, in terms of performance and accuracy, as well as motivating, in the second case, the stack trace lift map choice, among those suggested in 5.3.2.

6.1 Full Functions Tracking vs Network Stack Sampling

Previous chapters already conveyed that the Full Functions Tracking strategy would prove inadequate due to its increased CPU load on the system, thus requiring the definition of an alternative approach, which materialized in Network Stack Sampling. In this section, these allegations are finally backed up by suitable data, allowing the reader to come to an equivalent conclusion.

In order to assess the runtime performance implications of *Netto*'s monitoring features, the *iperf3* [3] throughput testing tool is used to produce a synthetic networking load between two identical Intel Core i7-6700 based machines running Ubuntu 22.04 LTS (kernel 5.15) and directly connected at 40 Gbps through a pair of *Intel XL710* QSFP+ NICs. Specifically, a single-stream TCP traffic was used as the

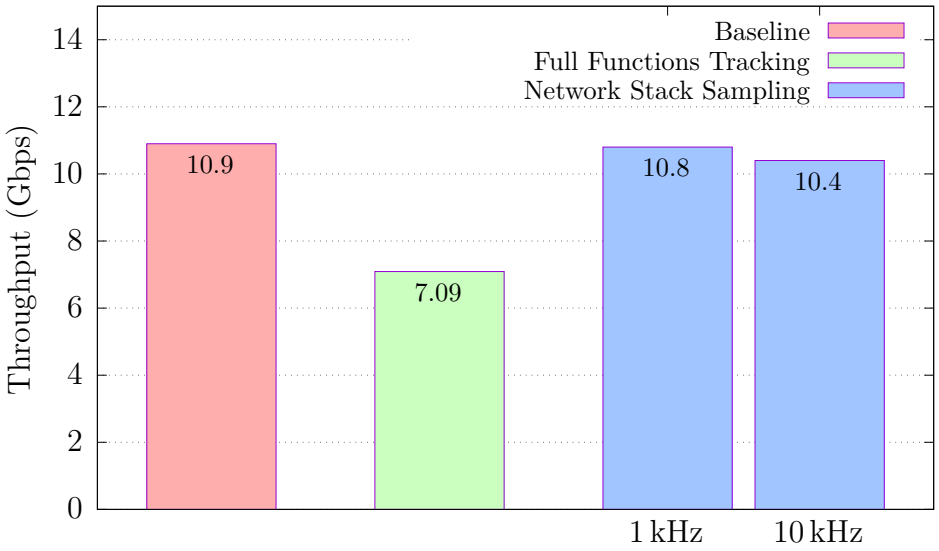


Figure 6.1: Comparison of the iperf3 TCP throughput between a baseline with no receiver instrumentation (red), Full Functions Tracking (green), and Network Stack Sampling (blue).

test-case, and *Netto* was loaded on the receiver (only the receiver needs to be monitored since the `NET_RX_SOFTIRQ` breakdown is a receive-side feature). Additionally, the *Generic Receive Offload* (GRO) capability has been disabled on the destination physical interface for two reasons: (*i*) a CPU bottleneck is introduced, bringing the practical throughput comfortably below the 40 Gbps link speed, and (*ii*) without any flow reconstruction feature, the rate of inbound logical packets is dramatically increased, aggravating the kernel load and better resembling a real-world scenario, where a typical server is unlikely to be handling a single huge TCP stream. With such configuration, *Netto*'s instrumentation overhead can be estimated implicitly via the imposed impact to the system's top throughput as reported by iperf3, instead of attempting difficult and possibly intrusive direct measurements. Figure 6.1 reports the results.

From the baseline of 10.9 Gbps measured without any form of runtime instrumentation, Network Stack Sampling only costs 100 to 500 Mbps depending on the configured sampling rate (1 and 10 kHz are represented in the figure), amounting to around 4.5% of the total in the worst case. Full Functions Tracking, instead, is responsible for a massive 35% performance degradation in the same conditions.

Furthermore, Full Functions Tracking's cost also depends on the type of load applied to the system, as different kinds and amounts of eBPF probes are hit. This is showcased by Figure 6.2, where the same iperf3 TCP receive test has been run with a special host configuration that puts the receiver NIC in a software bridge (light green). Compared to the previous results, adding an L2 bridge function to the

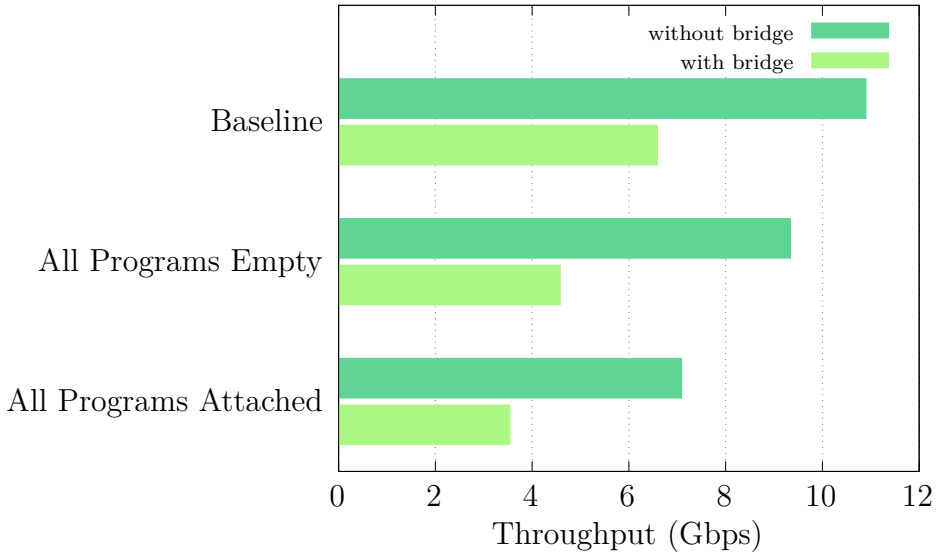


Figure 6.2: Overhead of Full Functions Tracking on the iperf3 TCP receive test under different networking configurations. When bridging is enabled on the host, effective speed is further reduced.

mix degrades performance significantly even in the "*Baseline*" case, where *Netto* is not loaded. When also considering the Full Functions Tracking capability (row "*All Programs Attached*" in the graph), the additional eBPF programs hooked to bridge-related code incur a further 10% performance penalty over the already high cost of the technique, bringing the total to about 45%: this figure suggests that Full Functions Tracking instrumentation can cost up to as much CPU time as the entire rest of the Linux network stack, in the worst case. The third row of the graph ("*All Programs Empty*") refers to a special build of Full Functions Tracking where all the eBPF tracing programs were emptied out but still attached to the usual hook points; the results show that, although lower, a substantial portion of the whole overhead is attributable to eBPF itself and not only to the cumbersome event stack mechanism explained in Section 5.3.1. Finally, it must be noted that Network Stack Sampling's overhead does not depend on the host configuration or load type.

Another important aspect that is worth considering for the comparison between Full Functions Tracking and Network Stack Sampling is the achieved accuracy of the sub-events metrics. The former will in fact provide an exact measurement, whereas Network Stack Sampling performs an estimate of the metrics based on a sampling frequency parameter. In Figure 6.3 the accuracy of the sampling strategy is analyzed at different frequencies; the graph refers to three iperf3 TCP receive tests conducted at, respectively, 100 Hz, 1 kHz and 10 kHz sampling frequency, and it plots — for each of these configurations — a histogram showing the measured CPU utilization for the IPv4 Local Delivery `NET_RX_SOFTIRQ` sub-event. To improve

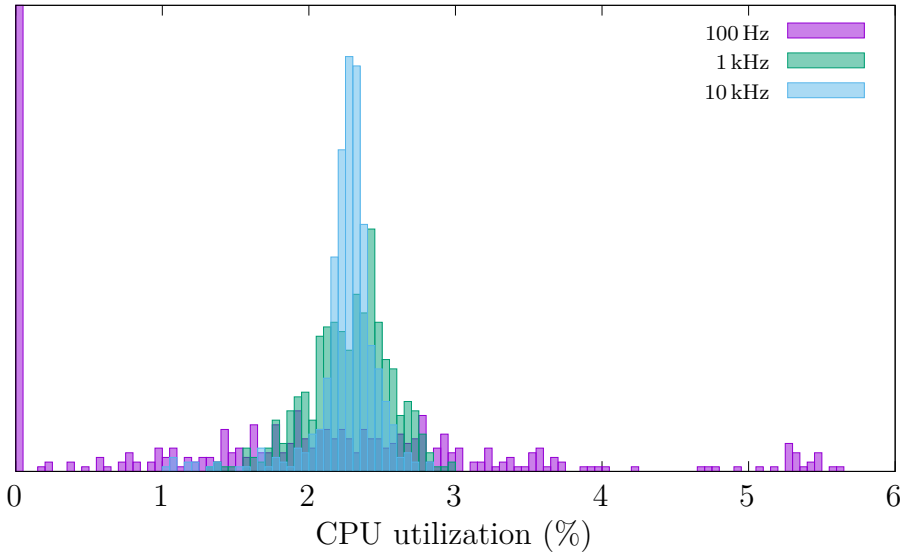


Figure 6.3: Relative accuracy of the measured IPv4 local delivery sub-event at varying sampling frequencies with the Network Stack Sampling breakdown method.

consistency in the measurement, the iperf3 bandwidth has been artificially limited to compensate for the increased load of the higher frequency runs, and all but one CPU cores of the system were disabled to account for scheduling and task migration noise.

The results show that at 100 Hz the reported values are unusable: the peak at $CPU_{utilization} = 0\%$ in fact conveys periods where the sampling pattern was so sparse that it missed all occurrences of the `ip_local_deliver()` function. Conversely, at both 1 and 10 kHz, the results appear promising, as only a marginal amount of accuracy is lost at the lower sampling frequency. Please note, however, that these tests heavily depend on the update period of the user-space side of the tool (here kept at the default $T = 500$ ms): higher values will smooth out the differences, possibly making lower frequencies more viable options. Furthermore, it would be incorrect to assume that the Full Functions Tracking method produces strictly better spreads, as its intrinsic overhead must also have a negative impact on the its accuracy.

6.1.1 Preferred Stack Trace Lifting Strategy

Having assessed the undoubted superiority of Network Stack Sampling over Full Functions Tracking, it is now appropriate to discuss any performance difference between the various proposed stack trace maps presented in Chapter 5. Among those four, the first solution based on a `BPF_MAP_TYPE_STACK_TRACE` is clearly far from ideal, due to its significant system overhead caused by the large amounts

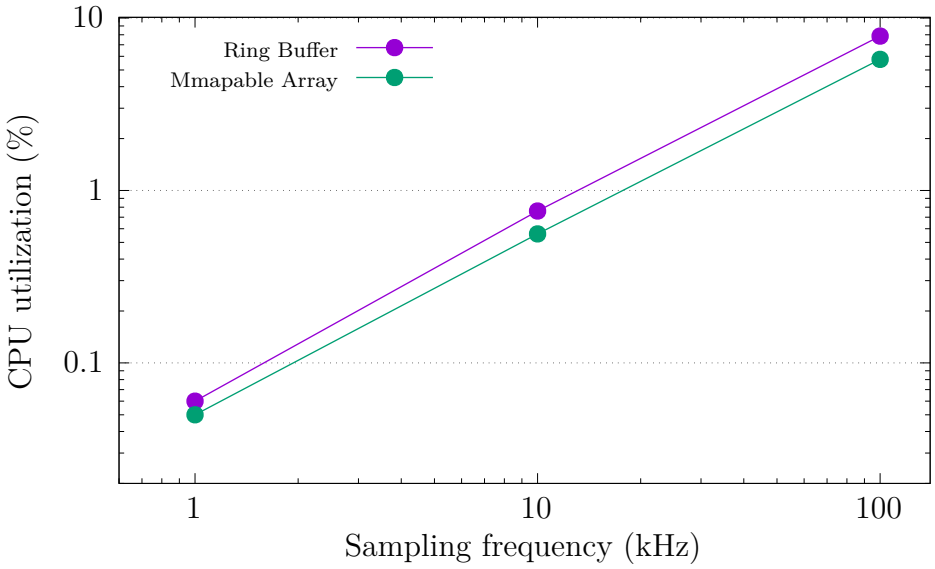


Figure 6.4: User-space CPU utilization for Network Stack Sampling with Ring Buffer and Mmapable Array backends.

of system calls required to retrieve data from it. Likewise, emulating the dedicated stack trace map with a hash table would not solve all of the problems, since memory would still need to be copied, potentially in the kernel context as well. For these reasons, the first two stack trace lifting strategies have not been implemented in code and have not thus been tested pragmatically. The other two, instead, are much more promising approaches, and picking a preferred implementation would not be a trivial task before testing their respective performance. This subsection hence tries to answer the question of whether the increased ease of use of the ring buffer comes with a performance overhead compared with the more barebones, lower level alternative.

In terms of networking throughput performance, the two solutions perform indistinguishably in all of the tests they have been subjected to. This suggests that the cost of submitting a ring buffer entry is virtually identical to that of writing to an array slot. Despite this performance similarity, the two implementations do not fare equally in other metrics. Figure 6.4 presents the average user-space cost (in terms of *unnormalized* CPU utilization) associated to both map types during an iperf3 receive test, at varying sampling frequencies.

This cost includes stack trace retrieval and analysis, as well as the calculation of the metrics. Although both approaches boast very low CPU overhead, the ring buffer abstraction consistently comes at the price of a slightly increased cost with respect to the lower level alternative. Also note that the sampling frequencies considered here are extremely high and only chosen to amplify the usually small cost of Network Stack Sampling; these values do not reflect the typical cost of a *Netto*

tracing session.

6.2 Controlled Tests

In this section, *Netto* is applied to various different scenarios in a set of controlled workloads, in order to showcase and analyze its results and further validate the output metrics. In all cases, the reference testbed is the one introduced in the previous section, composed of two identical Ubuntu 22.04 systems connected at 40 Gbps.

6.2.1 *iperf3* Throughput Tests

First off, the reliable iperf3 load generator was used to produce synthetic UDP and TCP traffic. This simple workload, while not representative of a typical networking load, is very useful for ensuring the correctness and demonstrating the quality of *Netto*'s diagnosis. Figure 6.5 shows, for four different transport protocols and flow direction combinations, a snapshot of the custom web frontend's real time overview graph, mentioned in the previous chapter.

In the top row, representing the TCP and UDP *send* tests, the results reflect prior expectations: the vast majority of the system's time is spent serving socket send operations, although a significant portion is also occupied by the ubiquitous `NET_RX_SOFTIRQ`, which in this case is mostly invoked to process acknowledgement packets (in the case of TCP, as demonstrated by the large "*Local delivery/v4*" representation on the rightmost bar), or for handling transmission completion interrupts, necessary for skb recycling logic and other tasks (which falls under the "*Driver poll*" category). Also notable is a thin sliver of cyan in the "kernel" bar of the TCP transmission graph, associated with the `NET_TX_SOFTIRQ` networking entry-point.

Moving over to the receive tests, the "TX syscalls" event has expectedly been replaced by its RX equivalent. The TCP test results are especially satisfactory as all of the kernel time is accounted for by networking tasks. Additionally, since *Generic Receive Offload* was enabled during the transfer, a major portion of the `NET_RX_SOFTIRQ`'s time is occupied by GRO overhead. Conversely, in the case of UDP, a large section of the kernel bar is reported as "*other*". To better understand this surprising anomaly, a kernel-side *flamegraph*[1] was generated by plotting the stack traces captured for Network Stack Sampling: Figure 6.6 shows the annotated result.

[1]Special kind of graph that can help visualize CPU activity on a per-function basis by plotting a set of stack traces.

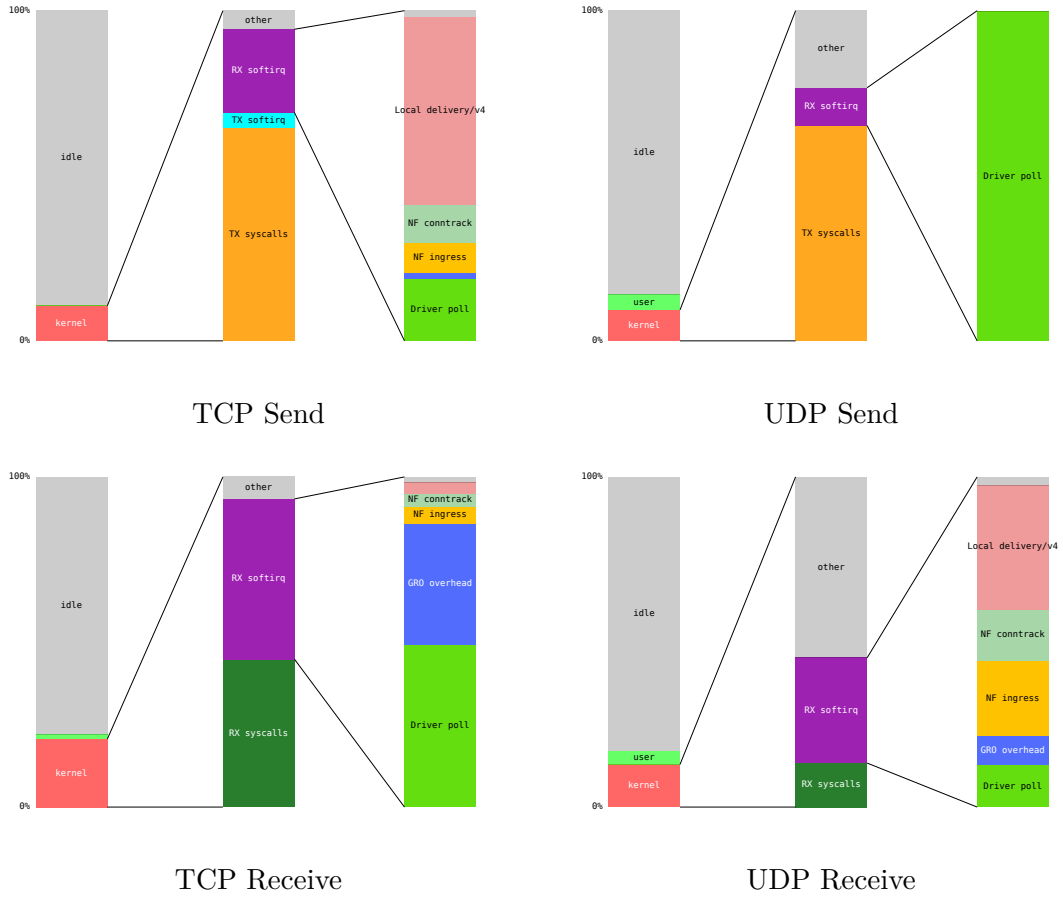


Figure 6.5: Measured networking cost during an iperf3 TCP (left) and UDP (right) send (top) and receive (bottom) tests.

The mysterious gap in the kernel's CPU time was thus revealed to be caused by other system calls — such as `select` — which were found to be used in iperf3's I/O loop. Furthermore, the generic syscall entry and exit overhead turned out to also represent a significant slice of the hidden cost: this workload was measured to generate about 700k system calls every second which, compared to the circa 7 times lower figure for equivalent TCP streams, makes context switch frequency a typical bottleneck for UDP transfers' throughput.

6.2.2 Google's *Online Boutique* microservices demo

To more closely analyze a typical web server workload, the Google's "*Online Boutique*" [4] microservices demo was also tested. The application, distributed as a set of eleven interacting microservices, was deployed on a *Kind* [13] Kubernetes cluster

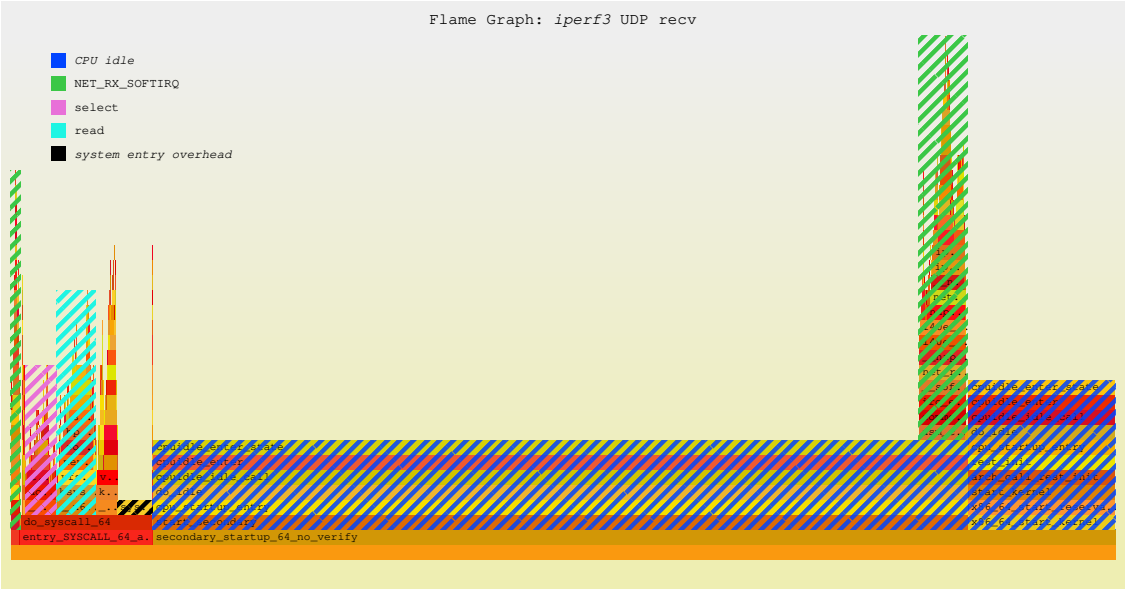
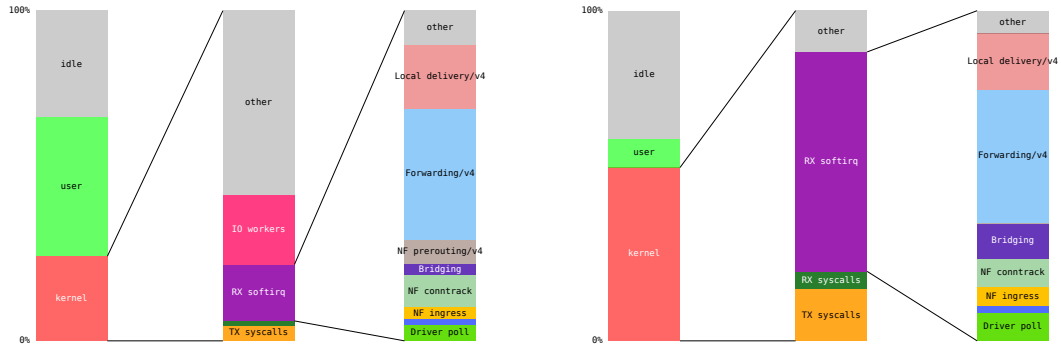


Figure 6.6: Flamegraph of the entire kernel's activity during an iperf3 UDP receive test.

and exposed through a *MetalLB* [17] load balancer. On the other machine, the additionally provided `loadgenerator` microservice — a *Locust*-based [1] application that can generate synthetic HTTP requests, and configured by Google to target multiple endpoints of the web service, thus creating load on several portions of the boutique deployment at the same time — was used to stress the cluster at a rate of about 1000 requests per second. As the test was ongoing, *Netto* captured the following CPU usage overview on the serving host (Figure 6.7a).

Immediately noticeable, as compared to the previous iperf3 tests, is the overall higher CPU utilization, which now comfortably exceeds 50%. This is easily explained by the fact that Locust can simulate the traffic generated by multiple users, whose requests can be handled independently; any competent web-server would thus spread its workload over different threads of execution, hence achieving better resource utilization on modern multi-core CPUs. In contrast, a single TCP or UDP transfer can not be handled by more than one thread, showcasing the lower processor usage. Additionally, the online boutique stress test has the server spend more time in user than in kernel mode. Again, this is not surprising as the tasks of application-level request parsing and handling, as well as the implementation of the business logic, all belong in the user-space domain. Regarding the time spent in the kernel, only a relatively small fraction of it can be attributed to raw networking work. Among this, most is due to the forwarding network function, responsible for exchanging data between the microservices in the host's internal virtual network. Another significant portion of the central bar is the "IO workers" section, which in



(a) Google "*Online Boutique*" (b) Memcached

Figure 6.7: Measured networking cost while stress-testing Google's "*Online Boutique*" (left) and Memcached (right).

this case maps to an `io_uring` SQPOLL thread: clearly, some microservice is using `io_uring` with the `IORING_SETUP_SQPOLL` flag set. The remaining kernel time is once again mostly spent in various system calls and relative system entry and exit overhead, as shown in the associated flamegraph (Figure 6.8).

6.2.3 *Memcached*

Memcached [16] is a high-performance generic distributed memory object caching system, frequently used in research literature as a benchmark for its fast packet processing and network stressing characteristics. For this thesis, Memcached has been deployed in a containerized form thanks to its official Docker image, and stressed by RedisLabs' `memtier_benchmark` (https://github.com/RedisLabs/memtier_benchmark), a Redis and Memcached traffic generator written in C++.

Once again, *Netto* was used on the Memcached server to capture its CPU utilization in real time, a snapshot of which is reported in Figure 6.7b. Compared with the microservices demo, the distributed object database showcases a far different allocation of the CPU resource between user and kernel modes, this time clearly favouring the latter. This behavior is a testament to the application's high level of optimization, as its performance is practically dictated by the kernel's network throughput. For this specific deployment option, a substantial portion of the networking cost is caused by bridging and forwarding, likely due to Docker's virtual network topology.

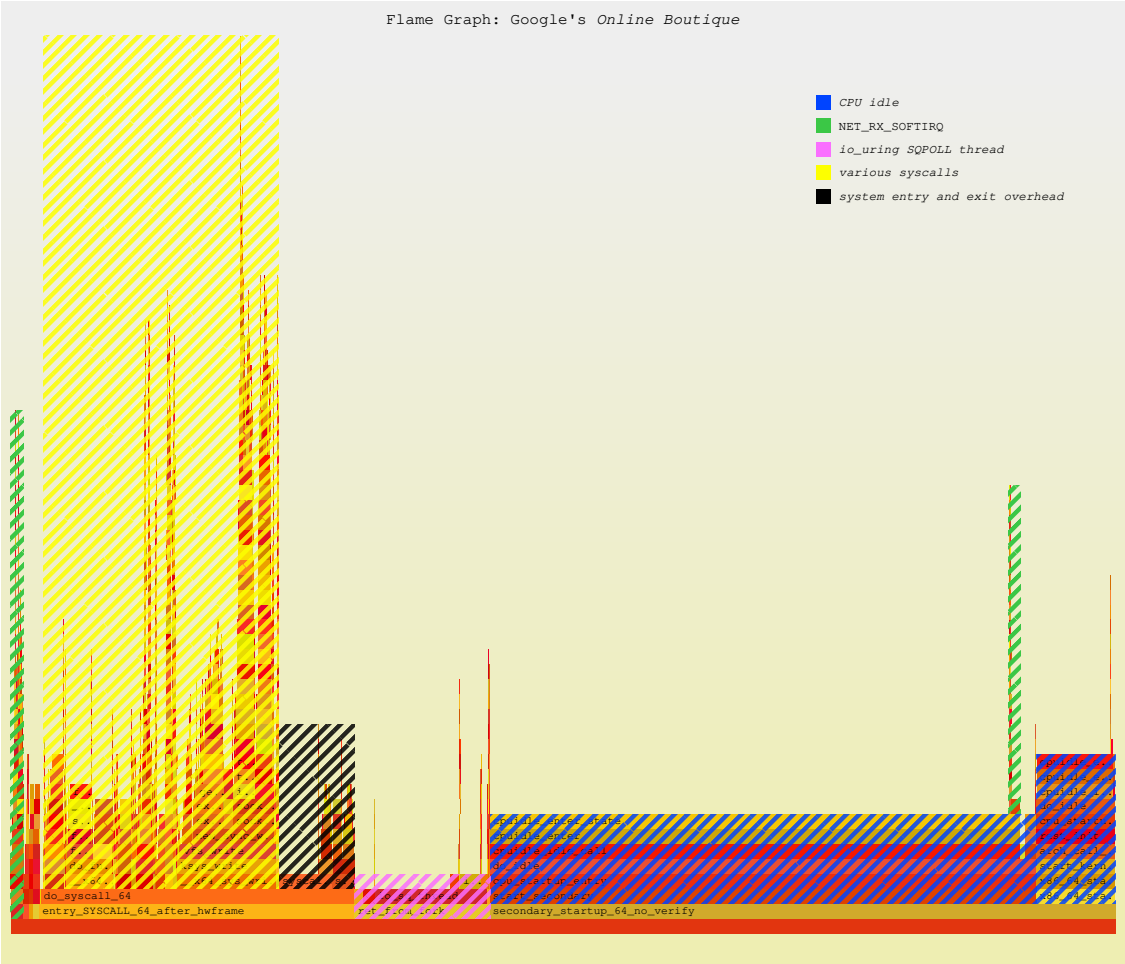


Figure 6.8: Flamegraph of the entire kernel's activity during the "*Online Boutique*" stress test.

6.3 CrownLabs Worker Monitoring

Finally, to test consistency and coherence of the measurements over extended periods of time, as well as verifying the tool on concrete production clusters, *Netto* was deployed on CrownLabs' (https://crownlabs.polito.it/) `worker4` node.

CrownLabs is Politecnico di Torino's internal computing cluster designed to provide students remote VM hosting for laboratories and exams. The system's infrastructure is based on a Kubernetes-powered cluster supported by a total of six dedicated servers; the KubeVirt project is then employed to support operating traditional virtual machines instead of relying on containerized desktop environment technology. The specific `worker4`, which also acts as the cluster's master, is powered by an Intel Xeon Gold 5120 14-core processor with SMT, for a total of 28 logical

Figure 6.9: CPU utilization statistics of the CrownLabs `worker4` node over a one-day observation period as captured by *Netto* and displayed by Grafana. The graphs show: overall CPU utilization (top), networking CPU utilization (middle) and breakdown of the `NET_RX_SOFTIRQ` (bottom).

CPUs.

To more comfortably track *Netto*'s metrics for this system over time, a Grafana-based toolchain was set up to display the time series and allow their exploration; additionally, the Grafana Mimir metrics database software was used for long-term data storage. These tools, along with the Prometheus scraper, were hosted on a second machine as to not clutter the CrownLabs node.

Figure 6.9 displays the results of a one-day long tracing session of the `worker4` node as displayed on a custom Grafana dashboard. The visualizations show (top to bottom): overall system CPU utilization, cumulative networking CPU utilization across the different network events, and the `NET_RX_SOFTIRQ` breakdown. As the data portrays, with an average system occupancy of about 20%, the Linux network stack is typically responsible for less than half a percentage point, except for the occasional peak caused by bursty data transfers. In any case, the nature of

this visualization allows system administrators to naturally see periodicities in the networking load, and debug possibly undesired behavior.

Chapter 7

Conclusions and Future Work

This thesis presented *Netto*, a Linux application that leverages the power and flexibility of the eBPF in-kernel virtual machine to perform dynamic tracing of an host's network stack in order to estimate the amount of CPU resources spent handling network packets. This information can in fact be of primary importance when evaluating a system's performance and can represent a helpful insight for enhancing efficiency and optimizing bottlenecks in a server's networking configuration.

During the design phase of the tool, the Linux network stack architecture was thoroughly studied to understand its structure and intricacies; four entry-points have thus been identified as the interface between the in-kernel networking services and the physical network or application domain: two dedicated softirqs and the socket send and receive operations. *Netto* hence relies on the instrumentation of these "*events*" with eBPF probes to compute each invocation's execution time, and then sum them together to obtain the overall CPU utilization of the entire network stack. As discussed, the choice of eBPF enabled *Netto* to provide consumption metrics in real time and with minimal system overhead, which itself allows continuous operation and integration with existing data center monitoring infrastructure. Additionally, the `NET_RX_SOFTIRQ` breakdown feature augments *Netto* with the ability to present more detailed information about the networking usage by showing dedicated metrics for most significant network functions such as bridging and forwarding, which can greatly improve the observability of the network stack's internals.

During the implementation stage, several challenges were faced due to synchronization concerns between the different tracked events, as well as unacceptable performance overhead induced by "*Full Functions Tracking*", the initial solution for the event breakdown logic, that therefore had to be later replaced by the improved "*Network Stack Sampling*" method, where a hybrid tracing/sampling architecture

solved the overhead problems while still maintaining adequate measurement accuracy.

Finally, in the evaluation step the tool's performance was analyzed, as well as discussing its results and reliability, by comparing its output on simple controlled tests with prior expectations and validating it with flamegraphs whenever such expectations were not met.

7.1 Current Limitations

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 DPDK-based custom data planes, as well as 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.

7.2 Future Work

The work presented for *Netto* could be reasonably expanded in several different directions. First off, the event breakdown capability could be expanded to consider additional and more specific sub-events, which could become significant for specific types of workloads not considered in this work. Besides, the breakdown functionality could also be extended to other top-level events beyond the `NET_RX_SOFTIRQ`, a possibility not yet contemplated by the tool. These considerations could give *Netto* a greater resolution for addressing and tackling networking costs.

Another aspect worth exploring in the future is the possibility to further reduce system overhead caused by the eBPF instrumentation. Although usually minimal for the networking and hardware configuration considered during testing, there's a possibility for higher impacts to be observed in different setups. In particular, faster link speeds could determine the increased CPU load for eBPF probing due to inflated event invocation frequency: at or above 100 Gbps, the default `NET_RX_SOFTIRQ` batching factor of 300 could prove insufficient to offset the received packet frequency, thus resulting in significant eBPF overhead. A possible solution to this issue could be implemented by moving the entire measurement stack to the sampling domain (which exhibits a fixed and user-configurable CPU

cost), that has already been shown to be capable of providing sufficient accuracy at minimal overhead.

Finally, a more ambitious goal would be to experiment with hooking *Netto*'s networking observation capabilities into third party projects, where the real-time networking utilization data could allow to dynamically improve system efficiency or performance, for instance by driving a data center's consolidation algorithms based on instantaneous CPU utilization metrics.

Bibliography

[1] Carl Byström et al. *Locust: open source load testing tool*. URL: https://locust.io/.

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

[3] Jon Dugan et al. *iperf3 throughput testing tool*. URL: https://iperf.fr/.

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

[5] 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.

[6] *Intel®64 and IA-32 Architectures Software Developer's Manual*. Accessed: 2023-09-01. Intel Corporation. 2023. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[7] *Kepler: Kubernetes-based Efficient Power Level Exporter*. Sustainable Computing. URL: https://github.com/sustainable-computing-io/kepler.

[8] 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.

[9] 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.

[10] *WebAssembly Core Specification*. Version 2.0. https://webassembly.github.io/spec/core/_download/WebAssembly.pdf. W3C, Apr. 19, 2022. URL: https://www.w3.org/TR/wasm-core-2/.

[11] Alexei Starovoitov. *BPF trampolines Linux patch*. URL: https://lore.kernel.org/bpf/20191114185720.1641606-5-ast@kernel.org/ (visited on 09/14/2023).

[12] The eBPF Community. *eBPF*. URL: https://ebpf.io/ (visited on 09/01/2023).

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

[14] The Linux kernel development community. *BTF: the eBpf Type Format*. URL: https://www.kernel.org/doc/html/next/bpf/btf.html (visited on 09/14/2023).

[15] The Linux kernel development community. *Linux NAPI*. URL: https://docs.kernel.org/networking/napi.html (visited on 09/15/2023).

[16] The Memcached contributors. *Memcached*. URL: https://memcached.org/.

[17] The MetalLB Contributors. *MetalLB*. URL: https://metallb.universe.tf/.

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