InXpect: Lightweight XDP Profiling

Vladimiro Paschali
University of Rome - Sapienza

Andrea Monterubbiano
CINECA

Francesco Fazzari
University of Rome - Sapienza

Michael Swift
University of Wisconsin—Madison

Salvatore Pontarelli
University of Rome - Sapienza

ABSTRACT

The eBPF eXpress Data Path (XDP) allows high-speed packet processing applications. Achieving high throughput requires careful design and profiling of XDP applications. However, existing profiling tools lack eBPF support. We introduce InXpect, a lightweight monitoring framework that profiles eBPF programs with fine granularity and minimal overhead, making it suitable for XDP-based in-production systems. We demonstrate how InXpect outperforms existing tools in profiling overhead and capabilities. InXpect is the first XDP/eBPF profiling system that provides real-time statistics streaming, enabling immediate detection of changes in program behavior.

CCS CONCEPTS

• **Networks** → **Network performance analysis**.

KEYWORDS

eBPF; XDP; performance; profiling

1 INTRODUCTION

In recent years, eBPF and XDP (eXpress Data Path) have emerged as powerful tools for executing a wide variety of network functions at high speed in the standard Linux kernel [16]. eBPF is used in performance-critical areas like firewalling [15], load balancing [10], and observability [11].

However, tools exploiting eBPF for tracing and observability of both kernel and user-level applications cannot be used to profile other eBPF programs. In other words, what is missing is a mechanism to enable introspection of eBPF applications for self-monitoring.

eBPF '25, September 8–11, 2025, Coimbra, Portugal

ACM ISBN 979-8-4007-2084-0/2025/09
https://doi.org/10.1145/3748355.3748367

One common method to investigate eBPF program performance is through external profiler tools such as the *Linux perf* suite [1] or *bpftool* [4]. *perf* is a powerful Linux performance analysis tool that can profile both user and kernel space, capturing hardware events like CPU cycles and cache misses. *bpftool* focuses on eBPF programs, inspecting execution statistics, map usage, and memory consumption. Unfortunately, using *perf* and *bpftool* has a significant overhead that can compromise the quality of the gathered results. Indeed, this overhead is way more detrimental when profiling XDP code, due to the performance requirements of networking applications in terms of throughput: as it will be shown later, throughput is reduced by up to 75% for simple XDP applications.

Moreover, existing tools do not allow users to examine specific functions or code sections, making identifying issues and performance bottlenecks difficult.

In this paper, we propose InXpect, a lightweight system for profiling XDP applications that minimizes profiling overhead and provides the flexibility needed to enable advanced profiling, such as fine-grained code block analysis.

Implementing a lightweight profiling system inside eBPF requires overcoming several challenges related to the eBPF execution model, which prevents the direct use of the underlying CPU hardware performance counters. To enable the direct use of performance counters in eBPF applications, InXpect exploits the recently added eBPF ability to call generic, user-defined kernel functions called *Kfunc*. We further reduce overhead through run-time sampling and careful access to eBPF array maps.

We tested InXpect with a set of XDP applications, obtaining a reduction in monitoring costs up to 8 times with respect to *perf*. Furthermore, to show our system's usability, we profiled a simple XDP traffic monitoring application, identifying a performance bottleneck and resolving it, gaining a 21% throughput in terms of processed packets.

Contributions. In this paper, we:

- design and implement InXpect, a lightweight system for profiling XDP applications;
- compare InXpect with state-of-the-art profiling tools;
- investigate the trade-off between accuracy and overhead, varying the sampling rate of InXpect;

- show how the use of our system can identify performance bottlenecks.

2 MOTIVATION

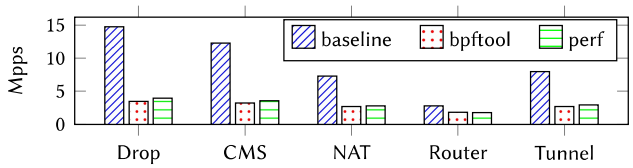

Figure 1: Throughput of different network functions with and without state-of-the-art profilers

The main critical aspects of widely used profilers such as *perf* and *bpftool* are: (i) the overhead they introduce, (ii) the inability to perform fine-grained profiling, and (iii) the lack of online monitoring, since collected results are available to the user only at the end of the profiling interval. In the following section, we delve into the details of these limitations.

Profiling Overhead. The main motivation driving our work was the excessive overhead (~600 instructions) introduced by *perf* and *bpftool*. Figure 1 clearly illustrates this overhead: we measured the throughput, in terms of millions of processed packets per second (Mpps), of several network functions executed by XDP with and without profiling tools attached. The results are clear: profiling causes a consistent throughput decrease for all the applications evaluated. This overhead prevents the chance of analyzing high-speed network functions in production [8].

Per-block profiling. *perf* and *bpftool* profile eBPF by: (i) executing a small eBPF snippet before the monitored program to read performance counters; (ii) running the profiled eBPF; and (iii) executing another eBPF snippet afterwards to read the counters again and store the difference. This allows the collection of statistics at the granularity level of the entire monitored program. In some cases, it could be of interest to collect measurements at a finer level to be able to identify more precisely the root cause of a performance issue (detectable by measuring the number of instructions, cache misses, etc...). For instance, a user may want to characterise the latency, computed in terms of clock cycles, of different code regions in order to identify the slowest one, or they may have to choose between two memory-intensive data structures and want to figure out which one incurs the smallest amount of memory accesses. With InXpect we aim to overcome this limitation by allowing users to define code sections to be explicitly analyzed and specify which statistics must be collected for each section.

Online Monitoring. Transient shifts in a program's behaviour can occur unexpectedly due to changes in traffic characteristics, placing stress on specific parts of the XDP code, exposing bottlenecks that were not evident during the design and optimisation phase. Traditional tools like *perf* and *bpftool* provide only cumulative statistics after detachment from the monitored eBPF program, offering limited insight into such sudden performance deviations. In contrast, a low-overhead online monitoring tool enables real-time analysis of collected statistics, making it possible to quickly detect these transient issues.

3 BACKGROUND

3.1 Performance Monitor Counters

Most CPUs currently include a set of counters inserted in their circuitry to gather information during code execution [12]. These counters monitor a wide set of events, such as the number of executed instructions, the number of hit/miss in caches, the branch mispredictions rate, and many others. State-of-the-art profilers such as Linux *perf* [13] and Intel V-Tune [6] are also based on the use of PMCs. A drawback of the PMC-based profiling is the additional overhead introduced [9]. This overhead depends on the profiling mode, the sampling frequency, and the techniques adopted to collect and read the gathered data. Furthermore, the use of profiling instruments could introduce side effects such as profile data perturbation, skewing results and deceiving the programmer.

Finally, generic profiling tools have been designed to cover a wide set of profiling scenarios. For example, since the same CPU can be shared among different tasks, a generic profiler must filter the events just for the task under analysis, discarding the others. This flexibility came at the cost of a complex and heavy infrastructure that made these tools unsuitable for profiling XDP applications.

3.2 Existing tools overhead

Starting from Linux kernel version 6.5, *perf* can attach its probes to BPF programs; the same is true for *bpftool* starting from version 7.4.0. Skimming through the source code of these tools, we identified the main function used to retrieve the performance counters' value and store it for later retrieval. Both *perf* and *bpftool* read the value of a counter twice, once at the beginning and once at the end of a program, by accessing a file descriptor stored into a *Perf Event Array*. A single access to this structure causes an overhead of around 370 CPU instructions. The overall overhead, comprising further access to an eBPF array, reaches around 760 instructions. This overhead is often too high to profile XDP applications correctly. since high-speed network functions usually execute a few hundred instructions for each processed packet.

4 CHALLENGES

InXpect profiles eBPF/XDP applications by inserting additional function calls in a program source code to read the PMC and send the collected information to the user space.

Since these functions will work at the kernel level and be executed by the eBPF virtual machine, several challenges must be solved to implement this self-monitoring approach efficiently.

Challenge #1: restrictions due to eBPF instruction set. The XDP applications are compiled using the eBPF instruction set. This choice allows easy verification of the code to be executed inside the kernel but denies the execution of native CPU architecture-specific instructions, such as those used to read and modify the PMC. We exploit the recently added capability of the so-called *Kfunc* [7] to supersede this limitation. A *Kfunc* is a kernel function that has been annotated and specifically designated as being callable from eBPF programs. They are an alternative to eBPF helper functions and provide an efficient method to add specific functionalities to eBPF programs. Since these functions are compiled with the native CPU instruction set, we can wrap the `rdpmc` x86 instruction used to read the PMCs in a kernel function exposed as a *Kfunc*.

Challenge #2: Low-overhead run-time instrumentation. Hardware performance counters are a powerful tool for application profiling. However, their usage can require significant instrumentation overhead. For example, *perf* access to the PMCs through an interface built around file descriptors. These file descriptors are allocated with the `perf_event_open()` system call, and counters are managed by using the `ioctl()` and `read()` system calls.This infrastructure is necessary for *perf* to profile userspace applications subject to task switching, hence requiring complex mechanisms to correctly access PMC values. Instead, since XDP applications are non-preemptible, all this effort is redundant, and we can safely access the PMCs directly, trimming the overhead significantly (see section 6.2).

Challenge #3: data collection. Collecting PMC information for each execution of XDP applications requires processing several million values per second. The collected counters must be sent to the userspace for processing, one-by-one or in an aggregated manner. For the first kind of approach, eBPF provides a *Ring Buffer* data structure, a multi-producer, single-consumer queue. However, this approach still incurs significant overhead due to the copying of every profiled event for each received packet Therefore, in InXpect we opted to aggregate the PMCs values inside the eBPF map type *Array Map*. In section 5 we show that, even in the the fastest eBPF application, we can read these map's values from userspace as soon as they are produced, effectively allowing online monitoring of the program's performances.

5 INXPECT

InXpect identifies the code regions to be profiled at compile time and executes a set-up phase before running the application under profiling. During the application execution,

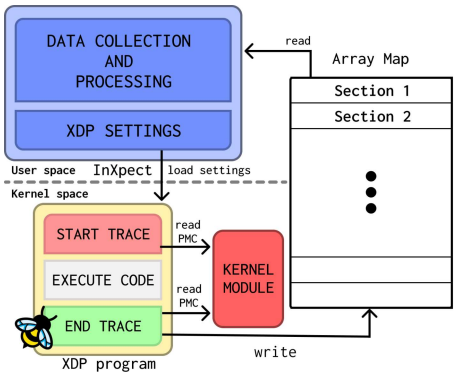

Figure 2: InXpect components

the user can monitor the program's behavior and modify the profiler configuration through the InXpect userspace component.

Identification of code regions. The programmer identifies the regions of interest in the source code and adds specific macros to delimit them, `START_TRACE` and `END_TRACE`. In the rest of the paper, we refer to code regions/sections as synonyms. Each region has a unique name to make them identifiable at runtime from the userspace component of InXpect. The `START_TRACE` macro adds the required code to read a PMC and to store its value in a local variable. It also handles region activation/deactivation and the sampling rate: the PMC reading is executed only if the number of runs of the XDP application modulo the sampling interval is zero.

Framework set-up. Before running the XDP application, InXpect loads a specific kernel module providing the function that wraps the x86 instructions to read the PMCs. Once the XDP program is loaded, it is possible to start InXpect userspace executable by specifying, as arguments, the name of the XDP program to monitor and other parameters, like the hardware metrics of interests and the sampling rate.

Run-time execution. At runtime, selected performance counters are read twice, once per each macro, delimiting a monitored region. The selection of the right counter happens inside `START_TRACE` macro, where an eBPF array, used for runtime configuration, is accessed. A counter index is retrieved and passed as an argument when calling the *Kfunc* that implements PMC reading. Calling and returning from a *Kfunc* adds a minimal overhead of only 40 instructions.

Sampling. To further reduce the instrumentation overhead, we introduced a sampling functionality, in which profiling is not performed for each packet but is done according to a chosen sampling frequency. This strategy cannot be directly implemented with *perf* or *bpftool*. *perf stat* collects and aggregates statistics for each packet, whereas *perf record*, while supporting sampling, doesn't allow targeting a single eBPF program. Furthermore, the *perf record* applies sampling globally, with all the hardware events concurrently sampled with

the same rate. In contrast, InXpect can be configured with different sampling rates for each event, providing a more flexible environment for lightweight profiling.

Data collection. The PMCs values collected during execution are stored in a specific data structure, which is queried from the userspace. This data structure is a global or per-CPU array map storing a struct composed of the following fields: (i) the section name; (ii) the aggregated value of the events counted during the entire profiling; (iii) total number of PMC readings; and (iv) the counter index, which is mapped inside the kernel module to a specific performance metric.

Interaction between userspace and profiled application. The userspace application is responsible for both configuring the profiling elements inside the eBPF code and retrieving profiling data. The configuration parameters are passed to the XDP program through some global variables defined in the eBPF code. They are accessed as an eBPF map from the user space while the XDP program reads them as standard variables with no overhead related to using eBPF helpers.

The main configuration parameters are the list of active monitoring sections, the performance counters to read in each region, and the sampling rate. The possibility of activate/deactivate specific profiling regions enables the mechanism of selective profiling, described in other works [14].

After the configuration of the InXpect parameters, the userspace application resets the eBPF map where profiling data is collected and starts to poll this structure for newly available data. As already hinted in section 4, we have two options: using a per-CPU array map or a global array map. Using per-CPU structures allows us to understand how an XDP program is performing on different cores and if there are some core-related specific behaviors. This granularity comes at the cost of using a syscall to read the per-CPU arrays. On the contrary, when using a global eBPF array map shared among all CPUs, InXpect is able to map the structure in its process address space (through `mmap()` system call) and read its values through a simple memory access. The reduced overhead with respect to the syscall allows for significantly faster polling rates.

6 EVALUATION

In this section, we report the experiments assessing the overhead of InXpect compared with other profiling tools, describing how the profiling overhead is split among the different components of a profiling system. Furthermore, we show how the sampling rate impacts the accuracy and overhead of InXpect. We conducted our evaluation on the CloudLab infrastructure [5] using two machines (sm110p) from the Wisconsin cluster equipped with an Intel Xeon Silver 4314 [2] processor and connected with 200/100 Gbps Mellanox ConnectX-6 Dx NICs [3]. The experiment profile is available on CloudLab[1], and the source code is publicly accessible on GitHub[2]. We tested our framework with five applications, summarized in Table 1 against a synthetic-trace composed of 10 Million UDP packets belonging to 2M different flows.

Table 1: Applications used for evaluation

Program	Description	Action
Drop	Drops each packet.	XDP_DROP
CMS	Count Min Sketch 4 by 2^{13}.	XDP_DROP
NAT	Network Address Translation.	XDP_TX
Router	IP Look up in a routing table.	XDP_TX
Tunnel	IP header encapsulation.	XDP_TX

6.1 Instrumentation Cost

First, we assessed how each profiling tool impacts the performance of network applications, measuring both achievable throughput and packet processing latency. We evaluated these metrics by activating the Linux kernel's `bpf_stats_enabled` configuration via `sysctl`, which collects aggregated runtime (in nanoseconds) and the total run count for each eBPF program. Activating this setting introduces approximately 20ns of overhead per call, but remains more efficient than instrumenting code to gather data through a one-entry array map.

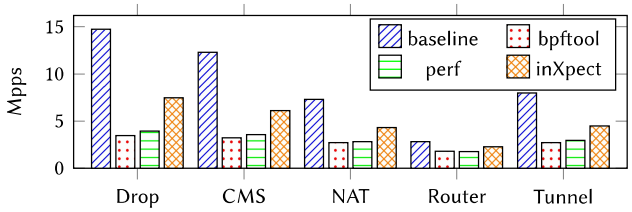

Figure 3: Throughput of the five eBPF applications used as benchmarks with various profilers

Figure 3 shows the throughput across our five benchmark applications and how it is affected when different profiling tools are attached. The baseline corresponds to the unprofiled application. All profilers are configured to count L1 cache misses. As expected, the Drop application suffers the most significant performance degradation: with just two instructions, the relative overhead of profiling becomes dominant.

Similar trends are observed across the other benchmarks, with *perf* and *bpftool* causing throughput reductions between 2x and 3.5x, while InXpect incurs a more moderate 1.2x to 2x slowdown.

Another interesting result comes from counting the average instructions executed using the examined profilers. Those numbers should be nearly identical since they are all attached to the same eBPF program. However, the results

[1]https://www.cloudlab.us/p/Metronome/InXpect
[2]https://github.com/VladimiroPaschali/eBPF-InXpect

shown in Figure 4, disprove this assumption. For the Drop application, for which we expect only 2 instructions, we count around 600 instructions using *perf*, 700 using *bpftool*, and 40 using InXpect; similar differences are found for the other applications. This effect is due to the profilers that count instructions belonging to the same profiling infrastructure. More specifically, all profilers execute some instructions before reading the performance counter for the second time. While this miscounting is quite small for InXpect, it is more significant for *perf* and *bpftool*.

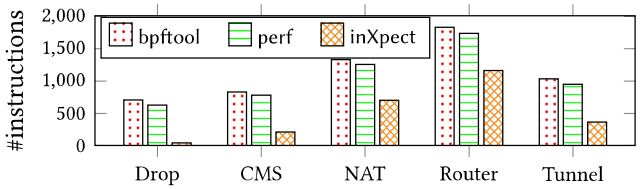

Figure 4: Instructions per packets of the five eBPF applications used as benchmarks with various profilers

6.2 Breakdown of Linux Perf overhead

We performed a specific test on the Linux *perf* tool to identify the different sources of its overhead. The same analysis holds for *bpftool*, since the two tools profile eBPF programs similarly. The main tasks executed by *perf* are:

- adding a *fentry()* and *fexit()* to wrap the eBPF code
- Call to *bpf_perf_event_read_value()* function to read the counter values

To obtain the breakdown of the profiling overhead, we used the instruction value computed for the naked Drop application as the baseline. We incrementally added to the application the different sources of overhead, deriving their specific burden by gathering the instruction count with *perf* and computing the difference with the baseline value. Results of this analysis are shown in Figure 5.

Attaching an empty *fentry()* / *fexit()* to an eBPF program costs around 400 instructions. These routines are used by *perf* to wrap the XDP/eBPF program under analysis. Inside *fentry()* / *fexit()*, a call to *bpf_perf_event_read_value()* is performed to read the PMC, which costs around 280 instructions. Additionally, the total profiling overhead also comprises a map update operation needed to store the difference between the two read counter values, but its 20 instructions are negligible with respect to the rest: this overhead is not reported in the graph for the sake of clarity. This breakdown explains the improvement provided by InXpect. As shown in Figure 5, in InXpect we completely avoided the use of the `fentry()` and an `fexit()`, since the profiling is enabled at compile time inside the very same profiled program. Moreover, *bpf_perf_event_read_value()* is substituted by a custom *kfunc*, reducing the overhead of InXpect to the two PMC read operations plus a negligible map update.

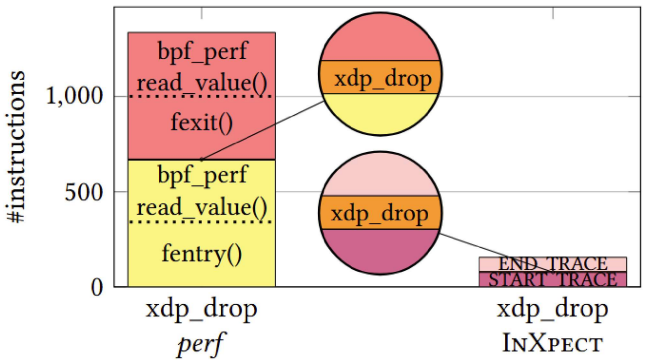

Figure 5: Breakdown of *Linux perf* (left) and InXpect (right) monitoring overhead. In the middle, we can see the zoomed instructions of the profiled application.

Perf Accuracy. In our experiments, we found that `fentry()` and `fexit()` functions not only introduce extra CPU instructions but also perturb PMC values collected via *perf*. We tested a modified Drop application, reading the packet header before dropping and counting L1 data-cache misses per packet. Our system correctly estimates about 1 miss per packet, but *perf* reports almost none, an odd result, as reading the header should cause at least one miss. Investigating further, we found that `fentry()` subtly interacts with the network driver. Normally, the driver prefetches the packet, but since the XDP program quickly accesses it, prefetching is usually ineffective. With `fentry()`, however, the added delay allows the driver to move the packet into the L1 cache before the header is accessed.[3]

6.3 Sampling rate trade-offs

While sampling effectively restores throughput close to that of an uninstrumented application, it is crucial to recognize its potential impact on the quality of collected metrics.

Figure 6 presents the packet throughput of our selected XDP applications, estimating L1 cache misses while varying the sampling interval between 8 and 1024 packets. L1 cache misses are selected due to their high sensitivity to perturbations and their relevance in diagnosing memory bottlenecks. Results for other metrics, though consistent, are omitted for brevity. The plots also report a baseline line, *i.e.* programs' throughput without profiling, and a "No sampling" line corresponding to profiling without sampling. To have a more comprehensive comparison between InXpect and *perf*, we implemented a sampling functionality within *perf*. This was done similarly to InXpect by implementing a run counter

[3] An error of one L1 data miss seems negligible, but for an XDP application that must process millions of packets, this effect can be significant. For example, in the CMS we target a miss rate of around 1 L1 miss per packet, and thus, the *perf* inaccuracy makes it really hard to optimize the CMS.

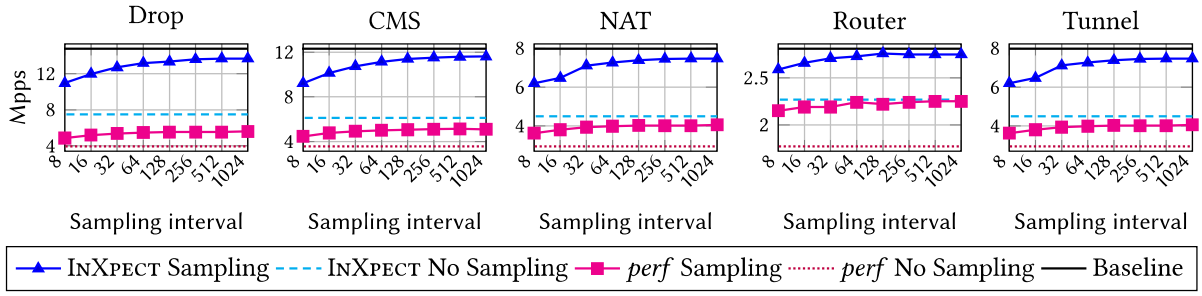

Figure 6: Packet throughput versus sampling interval with InXpect and *perf*.

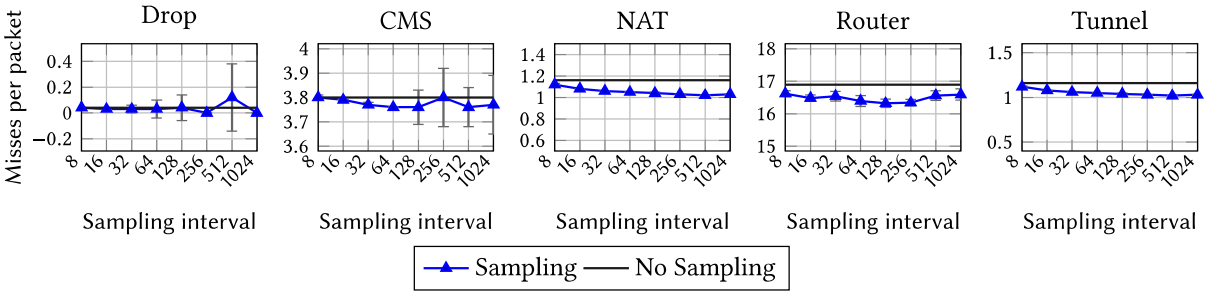

Figure 7: Packet cache misses versus sampling interval.

that is incremented at every program run and sampling if the counter is a multiple of the sampling interval. However, *perf* still has to call the `fentry()` for every program run, to check the sampling rate, which stops its performance improvements. For InXpect, the plots show how the profiling overhead decreases as the sampling interval rises. With a sampling interval of 16 packets, we obtain a throughput gain between 25% and 42%, depending on the profiled application. The data in Figure 7 presents the accuracy of gathered results when the sample interval varies. The experiment is repeated 10 times: plot lines depict the average values, with vertical bars showing the standard deviation. For the value range of interest, data shows a very small variation. The worst case is the Routing application with a variation of less than 10%, while it is almost negligible for other applications.

7 APPLICATION TO A REAL USE CASE

We used InXpect in order to analyze in detail the CMS application and improve its performance. Our Count Min Sketch parses the packet header, extracting the flow key; then, the flow key is hashed once for each row in the CMS, and the hash result determines the index of the array to be increased. We profiled the entire program, without any optimizations, and observed that it consumed about 281 instructions per packet, with a throughput of about 9.5 Mpps. Since InXpect allows the analysis of different program regions separately, we split the program into two sections: the first one handles packet header parsing, while the second updates the CMS data structure. We profiled two metrics: the instruction count and the L1 cache misses. The measured cache misses were in line with the expectations since we measured an average of 3-4 misses for a CMS of 4 rows. On the contrary, the number of instructions was quite high, around 240 in the second region. This was a hint that the overhead was caused not by a memory-intensive function but by a computation-intensive one. The most likely suspect was the hash function adopted, the Jenkins hash function. Hence, we decided to implement the more efficient DJB2 hash function. This optimization cut the instruction count from 281 to 176 and increased throughput to 11.5 MPPS.

8 CONCLUSION

This paper presented InXpect, a lightweight profiling tool for performance evaluation of XDP (and generic eBPF) applications. InXpect minimizes overhead by stripping unnecessary components from standard profilers like *perf*. It leverages hardware performance counters and eBPF *Kfunc* for efficient event tracking. InXpect allows selective, runtime-controlled profiling and includes statistical sampling to reduce overhead. Tested on XDP applications, it demonstrates real-world performance improvements.

9 ACKNOWLEDGMENTS

This work is supported by the European Union Commission's Horizon Europe, under grant agreements #101139282 and #101175702, and by the Sapienza University Grant ADAGIO (Bando per la ricerca di Ateneo 2023).

REFERENCES

[1] 2009. Performance counters for Linux. https://lwn.net/Articles/337493/.

[2] 2021. Intel Xeon Silver 4314 Processor. https://www.intel.com/content/www/us/en/products/sku/215269/intel-xeon-silver-4314-processor-24m-cache-2-40-ghz/specifications.html.

[3] 2022. Nvidia Mellanox ConnectX-6 Dx. https://www.nvidia.com/en-us/networking/ethernet/connectx-6-dx/.

[4] 2023. bpftool. https://bpftool.dev/.

[5] 2024. CloudLab. https://www.cloudlab.us/.

[6] 2024. Intel VTune Profiler Documentation. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler-documentation.html.

[7] 2024. KFuncs. https://ebpf-docs.dylanreimerink.nl/linux/concepts/kfuncs/.

[8] Marcelo Abranches, Oliver Michel, Eric Keller, and Stefan Schmid. 2021. Efficient Network Monitoring Applications in the Kernel with eBPF and XDP. In *2021 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. 28–34. https://doi.org/10.1109/NFV-SDN53031.2021.9665095

[9] Georgios Bitzes and Andrzej Nowak. 2014. The overhead of profiling using PMU hardware counters. https://openlab-archive-iv-v.web.cern.ch/openlab-archive-iv-v/sites/test-static-05.web.cern.ch/files/technical_documents/TheOverheadOfProfilingUsingPMUhardwareCounters.pdf. *CERN openlab report* (2014), 1–16.

[10] Facebook Engineering Blog. 2020. Katran: A Scalable eBPF-Based Load Balancer. https://engineering.fb.com.

[11] BPFtrace.org. 2020. BPFtrace Reference Guide. https://bpftrace.org.

[12] Stefano Carnà, Romolo Marotta, Alessandro Pellegrini, and Francesco Quaglia. 2023. Strategies and software support for the management of hardware performance counters. *Software: Practice and Experience* 53, 10 (2023), 1928–1957.

[13] Arnaldo Carvalho De Melo. 2010. The new Linux 'perf' tools. In *Slides from Linux Kongress*, Vol. 18. 1–42.

[14] Tanvir Ahmed Khan, Ian Neal, Gilles Pokam, Barzan Mozafari, and Baris Kasikci. 2021. Dmon: Efficient detection and correction of data locality problems using selective profiling. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. 163–181.

[15] LWN.net. 2019. BPFilter: eBPF-Based Firewalling. https://lwn.net.

[16] Marcos AM Vieira, Matheus S Castanho, Racyus DG Pacífico, Elerson RS Santos, Eduardo PM Câmara Júnior, and Luiz FM Vieira. 2020. Fast packet processing with eBPF and XDP: Concepts, code, challenges, and applications. *ACM Computing Surveys (CSUR)* 53, 1 (2020), 1–36.