Fast In-kernel Traffic Sketching in eBPF

Sebastiano Miano
Queen Mary University of London
s.miano@qmul.ac.uk

Xiaoqi Chen
Princeton University
xiaoqic@cs.princeton.edu

Ran Ben Basat
University College London
r.benbasat@cs.ucl.ac.uk

Gianni Antichi
Politecnico di Milano and
Queen Mary University of London
gianni.antichi@polimi.it

ABSTRACT

The extended Berkeley Packet Filter (eBPF) is an infrastructure that allows to dynamically load and run micro-programs directly in the Linux kernel without recompiling it.

In this work, we study how to develop high-performance network measurements in eBPF. We take sketches as case-study, given their ability to support a wide-range of tasks while providing low-memory footprint and accuracy guarantees. We implemented NitroSketch, the state-of-the-art sketch for user-space networking and show that best practices in user-space networking cannot be directly applied to eBPF, because of its different performance characteristics. By applying our lesson learned we improve its performance by 40% compared to a naive implementation.

CCS CONCEPTS

• **Networks** → **Network monitoring**; **Network measurement**; *Network performance analysis*;

KEYWORDS

Sketch, eBPF, XDP, Software Switch, Sketching Algorithm

1 INTRODUCTION

Virtualization is widely used in today's data center networks, where communication endpoints are no longer physical hosts but smaller individual units, such as Virtual Machines (VMs), containers, and serverless functions [2, 3, 18, 23, 33, 35]. A single physical server might host hundreds of VMs or thousands of containers, all connected to the network via a virtual switch [22, 46]. The virtual switch, therefore, plays a key role in ensuring the performance and security of the data center network [6]. In particular, it has become an important measurement vantage point, allowing network operators to gain a unique insight into inter-VM or inter-container network traffic that might not be visible at the physical switch level.

Past works [9, 28, 37] have explored enhancing user-space virtual switches with advanced measurement capabilities while maintaining high forwarding performance with kernel-bypass solutions such as DPDK. However, production-ready virtual switches (e.g., OpenVSwitch [46], Microsoft VFP [22]) still utilize the data-plane functionalities within the Linux kernel. This is because DPDK uses its own network device drivers, hence well-known tools for configuring, managing, and monitoring NICs (such as `tcpdump`, `ip link`, `ifconfig` etc.) do not work, making server management and debug hard [27]. Furthermore, with DPDK, administrators must maintain two separate networking configurations, one for the kernel and one for DPDK, increasing their management burden [49, 57]. Finally, in some deployment environments, it may not be feasible to dedicate the CPU and memory resources DPDK requires [54], because of the high per-core pricing of current deployments [56].

An emerging technology, the extended Berkeley Packet Filter (eBPF), enables the deployment of high-performance packet processing programs within the Linux network stack without the need to recompile it. By attaching an eBPF program to the eXpress Data Path (XDP) hook point, we can process a packet immediately when the kernel receives it from the NIC driver, before any subsequent connection-level and application-level processing. This allows to implement flexible data-plane logic while achieving high performance: an eBPF program can process as much as 20-25 million packets per second on a single CPU core [26, 27]. In fact, there are already several projects in industry [16] and academia [40] utilizing eBPF to implement data-plane functions for container networking.

This paper sheds light on the best practices for implementing high-performance measurement algorithms in eBPF. We focus our attention on sketches, as they provide rigorous accuracy guarantees and support a variety of measurement tasks [4, 42, 44], such as heavy hitters detection [9, 15, 51], per-flow frequency estimation [13, 17, 29], and counting distinct flows [5, 7, 38]. At a high-level, sketches are approximate data structures consisting of several counter arrays and a set of independent hash functions to update these counters. They are designed to reduce the memory usage of measurement tasks and to achieve guaranteed fidelity for the estimated statistics. Implementing a fast sketch under eBPF

requires a number of design decisions that open up several research questions: *(i) what is the best memory layout to use for storing the sketch?*; *(ii) what hash function provide the best trade-off between processing performance and collisions?*; *(iii) what is the overhead associated to a random number generator, needed by a sketch to perform probabilistic actions?* We show that due to the restrictions of eBPF and its instructions set, the answers to those questions cannot be directly taken from best practices in user-space networking (§2).

We implemented NitroSketch [37] in eBPF, the state-of-the-art software-based solution that builds on top of Count Sketch [14] and Univmon [38]. We then demonstrate that by applying the lessons we learned, we improved the performance of NitroSketch by 40% compared to a naive implementation (§3).

Contributions. In this paper we:

- Propose several optimization techniques that uniquely applies to algorithms running under eBPF (§2);
- Provide the implementation of NitroSketch, Count Sketch and Univmon, the last two needed by the first. We evaluate various optimization steps needed for achieving best performance (§3);
- Discuss how to close the performance gap between eBPF and user-space programs (§4);
- Share the code and all the benchmarks in open-source [39] (Appendix A).

2 ON EBPF PROGRAMS OPTIMIZATION

In this section, we discuss some optimization techniques we found unique for running sketches under eBPF. We also briefly summarize other generic optimization techniques that apply beyond eBPF.

2.1 The importance of `lookup` helper calls

Most sketches require a two-dimensional array of R rows and C columns which is then updated N times for every packet.[1] For user-space applications, the choice of implementing this (either as a matrix or a set of R 1-D arrays) does not have a huge impact on performance, as what it matters is just the number N of memory updates [37]. However, the eBPF environment requires different considerations. In eBPF, all stateful memories are implemented using *maps*, with *arrays* indexed by an integer being the simplest map type. Here, to implement a sketch, it is possible to lay out a $R \times C$ array in memory using four different methods:

- **Case #1**: An Array map with one entry containing the two-dimensional array. This needs, for every packet, one `map_lookup` call and N memory updates via direct pointer dereferencing.

[1]This number strictly depend on the specific sketching algorithm being used.

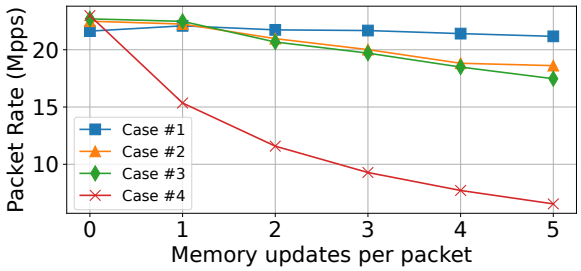

Figure 1: Reducing the number of `map_lookup` calls by changing memory layout can significantly improve the performance of eBPF programs.

- **Case #2**: An Array map with R entries, each of them containing an array of C elements. This imposes, for every packet, N `map_lookup` calls and N memory updates via pointer dereference.
- **Case #3**: An Array map with $R \times C$ elements. This requires, for every packet, N `map_lookup` calls and N memory updates via direct pointer dereference. Alternatively, N `map_update` calls.
- **Case #4**: An Array of Array map with R items where every item in the first-level map is a pointer to another map containing C elements. This needs, for every packet, N `map_lookup` calls on the *outer* map and N `map_update` calls on the *inner* maps.

It is worth noting that the amount of memory accesses for the first three cases is the same, while the number of map lookup calls differs. To test the performance implications of every solution, we implemented a common 5-rows sketch [37, 38] that writes up to 5 random locations for every received packet, one for each row, using *Global* (shared between CPUs) array storing 32-bit integers.

Our testbed includes two servers connected back-to-back with a dual-port Intel XL710 40Gbps NIC. The first, a 2x10-core Intel Xeon Silver 4210R CPU @2.40GHz with support for Intel's Data Direct I/O (DDIO) [1] and 27.5 MB of L3 cache, runs the various applications under consideration. The second, a 2x10 Intel Xeon Silver 4114 CPU @2.20GHz with 13.75MB of L3 cache, is used as packet generator. Both servers are installed with Ubuntu 20.04.2, with the former running kernel 5.13 and the latter kernel 4.15.0-112.

We found that the performance of eBPF programs is heavily affected by the number of `map_lookup` helper calls, which is a prerequisite for accessing persistent memory in eBPF. The impact of accessing a larger number of memory addresses in the same map is negligible, instead. Figure 1 shows the heavy performance penalty when the eBPF program needs to call the `map_lookup` helper function multiple times. However, when the eBPF program uses only one `map_lookup` call for multiple memory updates (case #3), its performance is

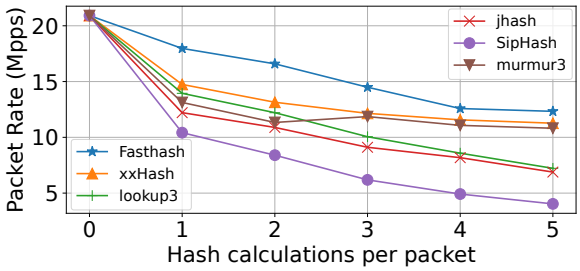

Figure 2: eBPF performance when calculating different number of hash functions per packet.

affected to a lesser degree by the increasing number of memory updates performed per packet. The significant performance impact of `map_lookup` calls is due to the extra cost for stashing register values when making a function call, as well as for the boundary checks performed due to memory safety.

Therefore, for any data structure stored in Array maps whose size is known at compile time, we should rearrange memory layout to use only a single map with one single entry, such that the eBPF program only make a single `map_lookup` call regardless of how many memory addresses are actually accessed.

2.2 The choice of the hash function

Calculating hash functions account for a considerable fraction of many network measurement algorithms' per-packet computational overhead. As observed by Liu et al. in NitroSketch [37], in a user-space software switch, the CPU spends as much as 37% of time calculating per-packet hash values. Thus, it is important to understand how different hash functions perform as a part of an eBPF program, and how they affect the packet processing rate.

We ran a single-core benchmark eBPF program that calculates hash values over each packet's 5-tuple (src/dst IP, IP protocol, and src/dst port numbers), and tests the packet processing rate when using different hash functions. We tested several hash functions: *xxHash*, the choice of NitroSketch authors; *jhash*, used by Linux kernel's hash tables; *fasthash*, and *lookup3*, two other fast hash algorithms. We also tested *SipHash*, a cryptographic secure hash function used by the Linux kernel, which might be required for applications processing adversarial traffic, and *murmurhash3*, the default hash function used by OVS [46].

We found that *fasthash* outperforms the other alternatives when running under eBPF (Figure 2), with an overhead that varies between ~16% when a single hash is calculated per packet, to ~40% with 5 hashes (required by 5-row sketch). This is surprising, since other hash functions such as *xxHash* and *murmurhash3* are often considered to be faster than *fasthash*, and indeed are the default choice in user-space projects.

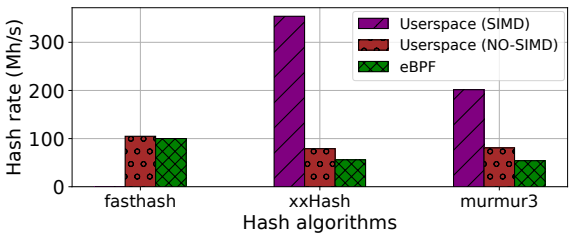

Figure 3: Performance comparison of different hash functions with SIMD instructions enabled and disabled (using Clang as compiler). eBPF cannot use vectorial instructions, as they are not allowed in the Linux kernel.

The impact of SIMD instructions. The first observation is that many modern hash functions like *xxHash*, or *murmurhash3* uses Single Instruction, Multiple Data (SIMD) in their high-performance implementation. Unfortunately, the eBPF instruction set does not provide SIMD capabilities.[2] In Figure 3, we benchmarked the performance of the vectorial version of *xxHash* and *murmurhash3*[3] hash functions when running as a normal user-space program with SIMD, and we compared it with the eBPF. In this case they performed the best in user-space, which is why they are chosen by prior works [37, 46]. However, their performance suffer when running under eBPF, and is outperformed by fasthash.

The importance of compiler optimizations. We tested the hash rate of the same set of hash functions shown in Figure 2 when independently benchmarked on the same CPU running in user-space without SIMD, using both GCC v11.1.0 and Clang v15.0. The results, shown in Figure 4, demonstrate the same trend when the program is compiled using Clang, which is the *default* compiler for eBPF targets. Surprisingly, *fasthash* performs a lot slower when compiled with GCC, the default compiler used by Nitrosketch. By looking at the generated assembly code, we noticed that Clang applies a more aggressive inlining compared to GCC, which in this case can provide considerable performance benefits.

The effect of the eBPF JIT compiler. We also noticed that the hash rate of the different hash functions under eBPF is always slower than the same version running in user-space. Although eBPF is Just-In-Time compiled to native instructions by the in kernel JIT, this performance deficiency can be explained by the limited instruction set available under eBPF. Here, all the hash functions, except *fasthash*,

[2]Currently, the use of SSE/AVX registers and instructions is highly discouraged in the kernel [53] for *(i)* portability problems, since not every architecture can support it, and *(ii)* for the additional cost of saving and restoring the FPU state.

[3]For the vectorial version of the hash functions we used the code available in [32]. For all the others we used the code available under the *SMHasher* [47] repository.

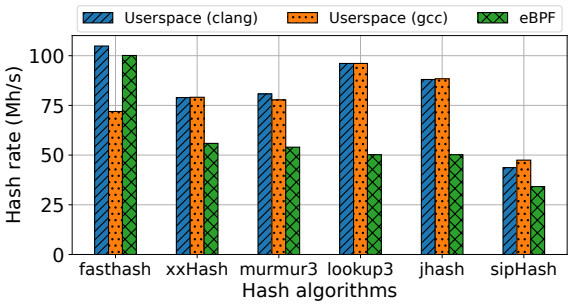

Figure 4: Performance benchmark of different hash functions with 16-byte input, when running in user-space vs. running under eBPF. For the user-space, we tested the same program compiled with GCC and Clang, while for eBPF Clang is the only choice.

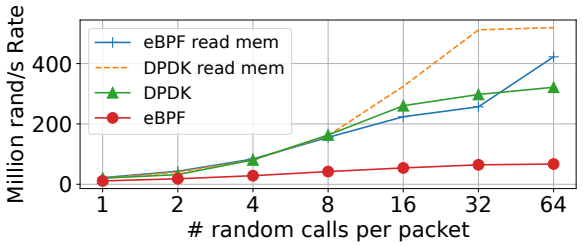

Figure 5: eBPF and DPDK performance when generating different number of random 32-bit numbers per packet using the same `prandom_u32` function. The eBPF/DPDK read mem plot indicate the performance when reading pre-generated random bits from memory.

use rotate instructions (e.g., `ROR`, `ROL`) that are included in the x86 instruction set. When the program is compiled to an eBPF target, those instructions are emulated with a set of *shifts* and *xor*, degrading the performance when running in the kernel.

2.3 Random bits generation

Random numbers are useful in many stochastic algorithms, including sketches. However, the canonical way of obtaining them (the `bpf_get_prandom_u32`) is costly [31]. We benchmarked the same function, `bpf_get_prandom_u32`, in user-space and eBPF by calling it an increasing amount of times for each packet (Figure 5). In eBPF, a single CPU core only produces approximately ~28 million 32-bit random numbers with 4 random calls per packet and ~67 Mrand/s with 64. In DPDK, we could generate ~81 million 32-bit numbers per second with 4 calls per packets, and ~320 Mrand/s with 64, ~5 times more than eBPF. The overhead for calling the random number generator in eBPF is given by the cost of external function calls for each individual 32-bit number; note that the penalty of register stashing for any external function call also applies here.

It is much faster for eBPF programs to read pre-generated random bits from map arrays that are populated by a user-space program. Here, the performance benefit is demonstrated in Figure 5, with the *eBPF read mem* line. Although it may sound obvious that loading pre-calculated numbers from memory can guarantee a performance boost against generating random numbers on-the-fly, we notice that a user-space program did not exhibit a performance gap when only generating 8 or fewer random numbers per packet. Meanwhile, the performance gap is considerably larger for eBPF programs. This is because every random number generated on-the-fly using `bpf_get_prandom_u32` incurs the extra cost of calling external helper functions, while we only need to make a single `bpf_map_lookup` helper call per packet when loading random numbers from memory, regardless of how many pre-calculated numbers we need to load.

2.4 Other generic optimization techniques

Finally, we note that the following optimization applies on any program that runs on modern CPUs, including those running as an eBPF program. Here, we discuss how to apply them in the context of implementing sketch algorithms for network measurement.

Reduce (unpredictable) branching. Branching hurts performance in today's pipelined CPU [30, 41]. In the context of network measurement programs, packet header parsing lead to branching; thus avoiding unnecessary parsing lead to reduced branching and improved performance.

Memory locality and cache residency. A predictable memory access pattern (locality) helps CPU pre-fetch data from the slower RAM into the faster L1/2/3 cache [12]. Unfortunately, sketch-based measurement algorithm has a random memory access pattern (due to using random hash functions for indexing). In this case, we should size the sketch accordingly such that the entire sketch to fit within the faster L1/2/3 cache.

3 CASE STUDY: OPTIMIZING NITROSKETCH

In this section, we share our experience implementing and optimizing a sketch-based measurement framework in eBPF.

NitroSketch is a measurement framework designed for running high-speed network measurement on CPUs, making it a good starting point for implementing high-speed sketching in eBPF. NitroSketch achieves high performance by reducing the number of memory accesses and hash calculations per packet. It is originally implemented as a part of popular software switches including OpenVSwitch-DPDK [48], VPP [8], and BESS [25].

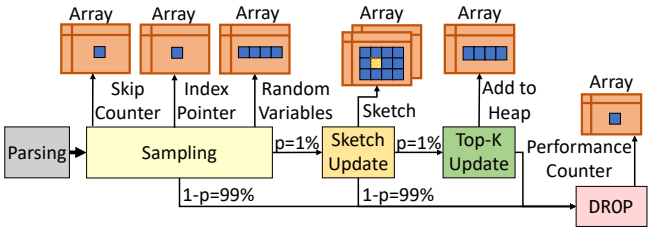

Figure 6: An unoptimized implementation of NitroSketch-UnivMon in eBPF, before consolidating all memory variables into a single array entry.

We implemented NitroSketch in eBPF with two underlying sketching algorithms, Count Sketch and UnivMon using approximately 200 and 300 lines of C code, respectively. Count Sketch [14] is used to produce unbiased estimation of flow sizes, while UnivMon [38] supports universal sketching, allowing many functions to be calculated over the frequency vector of flows. NitroSketch skips most sketch memory updates probabilistically while scaling up the remaining updates proportionally. For example, in a vanilla 5-row Count Sketch we need to add +1/-1 to five different locations in the sketch, one per row, while processing each packet. Building on top of it, in UnivMon there are many "layers" each hosting a Count Sketch and a heavy-hitters heap, and each new input packet updates all the rows in some of the layers. In NitroSketch-CountSketch, the update to each row is skipped with probability $1 - p$; otherwise (w.p. p), it adds $+1/p$ or $-1/p$ to the same location. The NitroSketch-UnivMon variant uses NitroSketch-CountSketch instead of vanilla ones in addition to updating at most one layer and using probabilistic heap updates. By choosing a small p, NitroSketch can achieve much higher packet-processing performance while not introducing a large impact to the sketch's accuracy. Also, for the same number of memory accesses, NitroSketch achieves better accuracy than uniformly sampling packets randomly and updating all rows.

We note that the original NitroSketch implementation supports dynamically changing the sampling probability p, adjusting it based on traffic throughput to maintain a constant CPU usage or achieve a certain accuracy guarantee. We can implement the same logic for eBPF by using a control plane script to change p dynamically for the data-plane eBPF program. In this paper, we benchmark using fixed p ranging from 1% to 100% to highlight performance improvements.

3.1 An eBPF implementation

In Figure 6, we illustrate how we implemented NitroSketch as an eBPF program. Our eBPF program has four components: Parsing, Sampling, Sketch, and Top-K. When the kernel first receives a packet and starts the eBPF program, we perform basic parsing to extract the flow ID 5-tuple (IP address pair, protocol, and port number pair) from the packet header. Subsequently, we execute the random sampling process and skip most of the packets. The eBPF program saves the number of packets to skip in a single-element map array. For most packets, we subtract this counter and move on; when the counter reaches zero, we update the sketch, and replenish the counter by fetching another pre-computed Geometric Random Variable[4] from the memory. This approach has lower performance overhead than calling the random number generator when processing every packet, as discussed in the original NitroSketch paper [37] and benchmarked in Section 2. For packets not skipped, we calculate hash functions over the packet's flow ID, to identify the row and column index and update the sketch value. For NitroSketch-UnivMon, we have two additional steps, which are subjected to a couple of limitation of the eBPF environment.

Lack of bit counting instruction on eBPF ISA. Before updating the sketch, in NitroSketch-UnivMon we need to select the layer to update. This is done by calculating an additional hash on the packet's flow ID and counting the number of trailing bit set. On a user-space program, this can be easily done by using a bit counting instruction provided as hardware operator by modern CPUs (e.g., `ctz`, `ntz`). However, the eBPF ISA doesn't support those instructions; we then opted for a software emulation of the `ntz` instruction, carefully chosen to exploit the parallelism of modern CPUs.

Lack of heap map in eBPF. When we run random sampling again to skip most packets, only a p fraction of them reaches the top-K update step, where we query its flow size and update a heap to maintain the top-K flows candidates. Since eBPF does not have an *heap* map, we emulated its behavior using a *sorted* array of size K that contains the flow ID and the flow size obtained after querying the sketch. When a packet reaches the Top-K update phase, we first check if the flow ID is present in the array and update the flow size with the new one; otherwise, we insert the new flow ID in the last position of the array, where the element with the smallest flow size is present. After the insertion, we run the *insertion sort* algorithm to build the final sorted array, which represents our heap. Finally, all packets are dropped after updating a performance counter, which counts the number of packets processed per second.

[4] A Geometric Random Variable is a variable that follows a geometric distribution [21, 34], which gives the probability that the first occurrence of success of a given event requires k independent trials, each with success probability p.

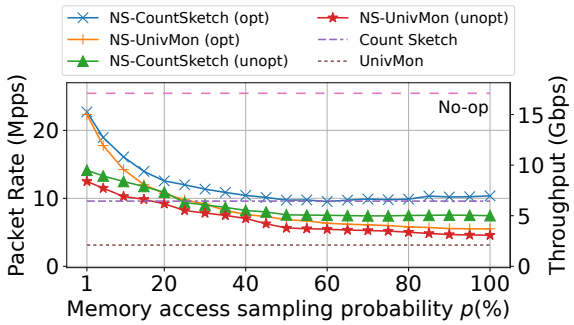

Figure 7: After extensive optimizations, our eBPF program achieves ~22.27 Mpps on one CPU core with 64B packets, for NitroSketch-CS and NitroSketch-UM at $p = 1\%$.

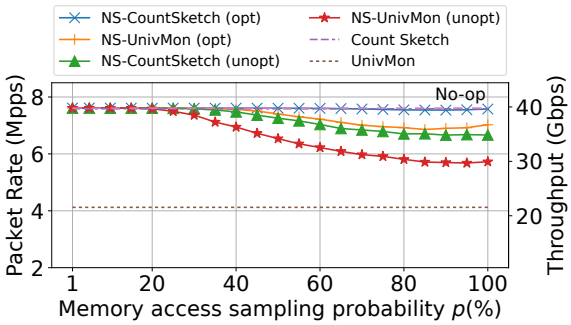

Figure 8: Single core packet rate (left y-axis) and throughput (right y-axis) of our different sketch implementations with a real-world Datacenter trace UNI1 from [10].

3.2 Optimizing NitroSketch in eBPF

Here, we share our experience in optimizing NitroSketch to achieve higher performance. Our initial NitroSketch implementation could process 5~10 million packets per second on a single CPU core. In contrast, a no-op eBPF program that just drops packets can process approximately 25 million packets per second on our testbed.

Workloads. We tested our implementations using two different workloads: (a) a simulated traffic trace with min-sized packets (64 bytes) for stress testing, with random destination IP addresses and randomized port numbers for a total of ~1 million distinct flows (Figure 7); (b) a data center trace [10], which has an average packet size of 542 bytes (Figure 8). We used `pktgen` [19] with DPDK v20.11.0 to generate random 64B packets, the DPDK *burst-replay* [20] tool to replay the trace and Receiver Side Scaling (RSS) queues to control the number of CPU cores used to run the eBPF program.

When we choose a small sampling probability (i.e., $p = 1\%$), the cost of sketch updates is negligible, and most overhead comes from skipping packets in the random sampling step. Here, we used the batched skipping technique discussed in the original NitroSketch paper [37], that pre-calculates Geometric Random Variables to skip consecutive packets, instead of generating random numbers per packet. Subsequently, we fixed $p = 1\%$ and analyzed our eBPF program's overhead, thus identifying four optimizations:

Optimization #1: ***Faster hash function.*** The original NitroSketch implementation uses `xxHash`, which can utilize SIMD instructions when running in user-space. However, the eBPF instruction set does not support SIMD, and we found `fasthash` runs faster.

Optimization #2: ***Fewer hash function calls.*** Instead of calculating a dedicated hash function for each of the four sketch rows, we split a single 64-bit hash value into four 16-bit parts, which are sufficient for indexing up to 2^{16} columns in the sketch.

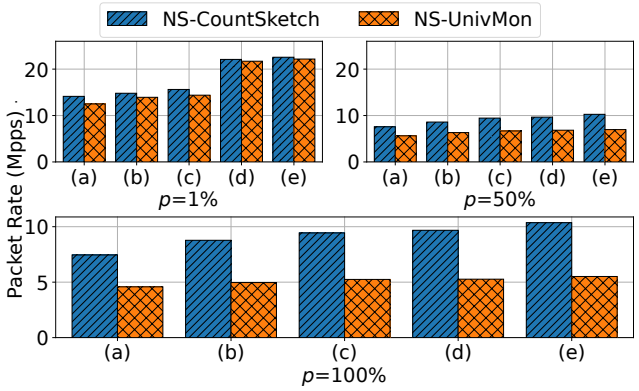

Figure 9: Breakdown of each optimization step we performed: (a) Unoptimized, (b) Use faster hash function, (c) Reduce hash calculations, (d) Postpone parsing after sampling, and (e) Consolidate array lookups.

Optimization #3: ***Swap parsing and sampling.*** Extracting flow ID 5-tuple requires branching, which in turn causes overhead due to CPU branch prediction (and misprediction). We reduced branching by skipping packets early and postponing unnecessary packet parsing, reducing from 8 branches per packet to 2.

Optimization #4: ***Consolidate array lookups.*** Instead of using several different map arrays and calling `map_lookup` multiple times, we combined all frequently-accessed memory variables (counters and geometric random variables) into one large struct in a single map array. This way, we only need to call `map_lookup` once per packet to access all these variables in the memory.

With these optimizations, we improved our implementation's performance by 15%~59%, and allow NitroSketch-CountSketch and NitroSketch-UnivMon to process on a single core ~22.27 Mpps with $p = 1\%$, compared to ~10 Mpps of the unoptimized version (Figure 7). This is near 90% of the maximum possible per-core packet processing rate on

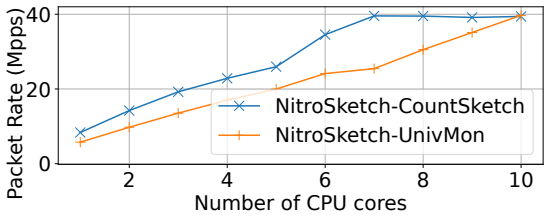

Figure 10: Packet rate (Mpps) with synthetic trace composed of 64B packets. Both NitroSketch-CountSketch and NitroSketch-UnivMon run with $p = 50\%$. The performance scales almost linearly when we use multiple CPU cores.

our hardware (25.5Mpps) when running a no-op eBPF program. In Figure 9, we present a breakdown of performance improvement after applying each of the aforementioned optimizations. With a low sampling probability $p = 1\%$, the most significant improvement comes from batched skipping packets and postponed packet parsing, while for a high sampling probability $p = 50\%$, and $p = 100\%$, the benefit comes from choosing the right hash function and calculating at most one hash per packet.

3.3 Scaling to multiple cores

Finally, we check whether the single-core performance shown earlier can scale in the presence of multiple CPU cores. We used the synthetic trace discussed above with minimum-sized packets and configured the number of CPU cores processing XDP by setting Receive-Side Scaling (RSS) queues on the NIC. We changed the number of RSS queues and show the total performance of our NitroSketch eBPF programs, with $p = 50\%$ (Figure 10). Here, the performance scales almost linearly with the number of cores until we reach a system-wide bottleneck (physical link rate limit).

4 DISCUSSION

4.1 Applicability of optimization techniques

Our benchmarks aim at improving network measurement algorithms in eBPF. Nevertheless, we believe our findings can be applied to a broad range of eBPF-based programs. For instance, we noticed that the Facebook's open-source L4 eBPF/XDP *load balancer* Katran [50] uses `jhash` as default hashing algorithm. This hash is computed on every incoming packet and it is used for connection tracking. As shown in Figure 2, this may result in an additional overhead compared to `fasthash`, the fastest hash algorithm in our benchmarks. We also noticed that Katran uses several `bpf_map_lookup` helper calls to retrieve configuration data and to save statistics about the running eBPF program (e.g., packets processed, dropped, new connections, VIP misses). If consolidated into a single large map element, the performance could increase, as shown in Section 2.1.

Similarly, we noticed that Rakelimit [45], the Cloudflare's multi-dimensional eBPF-based rate limiter, when deciding whether to accept or drop a packet, uses several `bpf_get_prandom_u32` calls and a combination of `fasthash` and `lookup3` hash functions to calculate the counter to update into a count-min sketch. Here, according to the lessons learned in Section 2.2 and the optimization #2 applied in Section 3.2, only using `fasthash` and splitting the 64-bit hash value into four 16-bit parts can improve performance.

4.2 Closing the performance gap

We believe the observed performance gap between eBPF and user-space programs can be reduced by applying various changes to the eBPF ecosystem. Here, we discuss possible next steps.

SIMD. The eBPF instruction set aims at achieving a delicate balance between expressiveness and cross-platform compatibility, as the Just-In-Time compiler efficiently compile eBPF instructions into native instructions on many platforms. SIMD is supported by many CPUs on the two most common architectures (amd64 and arm64). However, if SIMD instructions are added into eBPF instructions, the JIT compiler on other CPUs not supporting SIMD must "emulate" these instructions, possibly with minor performance penalty.

Penalty for helper calls. As per Section 2, calling eBPF helper functions leads to a significant performance penalty, likely due to register variables being stashed prior to making any such calls, similar to when a user-space program performs a syscall. However, it is possible to avoid the penalty for function calls. Calls into simple functions can be inlined during compilation, which eliminates the need for stashing. Furthermore, user-space high-performance network programs (e.g., DPDK-based programs) can be linked statically and benefit from link-time optimization (LTO). It might also be possible for the eBPF JIT compiler to perform an optimization similar to inlining/LTO, such that the "call" instruction to simple helper procedures can be translated directly into the body of the helper function.

Randomness pool. It may be possible for the kernel to generate pseudo-random numbers in batches and directly read from a pool of numbers upon every `bpf_get_prandom_u32` call, to achieve higher eBPF packet-processing performance similar to those achieved by manually pre-generating random numbers.

5 RELATED WORK

eBPF is a nascent field for host-based network research, with many ongoing works using eBPF to implement a high-performance and feature-rich data planes. Nevertheless, only

a few research projects target network measurements using eBPF. Otten and Bauer at Cloudflare presented RakeLimit [45], a prototype using eBPF to track hierarchical heavy-hitter flows and rate-limit them. Meanwhile, Bertin also discussed an eBPF-based DDoS defense system [11] deployed at CloudFlare. NetFlix built an eBPF-based network monitoring system [52] to produce flow logs. ViperProbe [36] is an eBPF-based microservice monitoring system that logs various performance metrics, including TCP send/receive bytes, retransmissions, and drops. Although eBPF is increasingly popular as a tool to benchmark other applications' performance [24], there are not many prior work investigating the performance of eBPF programs themselves. Jones [31] presented a detailed analysis on the performance penalty of various eBPF functionalities. In particular, they observed accessing arrays and clock timestamps are expensive, while generating pseudo-random numbers are cheaper. We complement their work by analyzing the performance of sketch-related functionalities in detail, and optimizing a complete sketch algorithm.

This paper focuses on eBPF programs running on x86 CPUs; however, it is possible to offload eBPF programs to run directly within NICs [26]. For example, Netronome SmartNICs [43] allows offloading eBPF programs, and Vega et al. [55] has demonstrated offloading packet-filtering eBPF programs onto an FPGA chip. We note that optimizing eBPF programs for running on FPGA/SmartNICs might require slightly different heuristics, and leave these as future work.

6 CONCLUSION

In this paper, we discussed the best practices for implement high-performance network measurement in eBPF, using sketch-based algorithms as case study. Optimization techniques commonly used in user-space programs, such as reducing branching and minimizing memory access per packet, also applies for eBPF programs. Surprisingly, we find restructuring the memory layout by using one large struct in a single map array (instead of multiple map arrays) significantly improve performance. To further improve performance, we can carefully choose the best performing hash function under eBPF and fetch pre-generated random bits from memory.

Acknowledgements. We thank the CCR reviewers for their insightful comments and suggestions. This work was funded in part by the 0UK EPSRC project EP/T007206/1 and by the European Union under the Italian National Recovery and Resilience Plan (NRRP) of NextGenerationEU, partnership on "Telecommunications of the Future" (PE00000001 - program "RESTART").

REFERENCES

[1] 2021. Intel Data Direct I/O Technology. https://www.intel.co.uk/content/www/uk/en/io/data-direct-i-o-technology.html. (Feb 2021). [Online; accessed 07-March-2023].

[2] Paarijaat Aditya, Istemi Ekin Akkus, Andre Beck, Ruichuan Chen, Volker Hilt, Ivica Rimac, Klaus Satzke, and Manuel Stein. 2019. Will serverless computing revolutionize NFV? *Proc. IEEE* 107, 4 (2019), 667–678.

[3] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*. 419–434.

[4] Anup Agarwal, Zaoxing Liu, and Srinivasan Seshan. 2022. HeteroSketch: Coordinating Network-wide Monitoring in Heterogeneous and Dynamic Networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 719–741. https://www.usenix.org/conference/nsdi22/presentation/agarwal

[5] Eran Assaf, Ran Ben Basat, Gil Einziger, and Roy Friedman. 2018. Pay for a sliding bloom filter and get counting, distinct elements, and entropy for free. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2204–2212.

[6] Siamak Azodolmolky, Philipp Wieder, and Ramin Yahyapour. 2013. SDN-based cloud computing networking. In *2013 15th International Conference on Transparent Optical Networks (ICTON)*. IEEE, 1–4.

[7] Ziv Bar-Yossef, TS Jayram, Ravi Kumar, D Sivakumar, and Luca Trevisan. 2002. Counting distinct elements in a data stream. In *International Workshop on Randomization and Approximation Techniques in Computer Science*. Springer, 1–10.

[8] David Barach, Leonardo Linguaglossa, Damjan Marion, Pierre Pfister, Salvatore Pontarelli, and Dario Rossi. 2018. High-speed software data plane via vectorized packet processing. *IEEE Communications Magazine* 56, 12 (2018), 97–103.

[9] Ran Ben Basat, Gil Einziger, Roy Friedman, Marcelo C. Luizelli, and Erez Waisbard. 2017. Constant Time Updates in Hierarchical Heavy Hitters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 127–140. https://doi.org/10.1145/3098822.3098832

[10] Theophilus Benson, Aditya Akella, and David A. Maltz. 2010. Network Traffic Characteristics of Data Centers in the Wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement (IMC '10)*. Association for Computing Machinery, New York, NY, USA, 267–280. https://doi.org/10.1145/1879141.1879175

[11] Gilberto Bertin. 2017. XDP in practice: integrating XDP into our DDoS mitigation pipeline. In *Technical Conference on Linux Networking, Netdev*, Vol. 2.

[12] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. 2021. Understanding Host Network Stack Overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 65–77. https://doi.org/10.1145/3452296.3472888

[13] Moses Charikar, Kevin Chen, and Martin Farach-Colton. 2002. Finding Frequent Items in Data Streams. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming (ICALP '02)*. Springer-Verlag, Berlin, Heidelberg, 693–703.

[14] Moses Charikar, Kevin Chen, and Martin Farach-Colton. 2004. Finding frequent items in data streams. *Theoretical Computer Science* 312, 1 (2004), 3–15.

[15] Kenjiro Cho. 2017. Recursive Lattice Search: Hierarchical Heavy Hitters Revisited. In *Proceedings of the 2017 Internet Measurement Conference (IMC '17)*. Association for Computing Machinery, New York, NY, USA, 283–289. https://doi.org/10.1145/3131365.3131377

[16] Cilium. [n. d.]. eBPF-based Networking, Observability, and Security. https://cilium.io/. ([n. d.]).

[17] Graham Cormode and Shan Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.

[18] Cosmin Costache, Octavian Machidon, Adrian Mladin, Florin Sandu, and Razvan Bocu. 2014. Software-defined networking of linux containers. In *2014 RoEduNet Conference 13th Edition: Networking in Education and Research Joint Event RENAM 8th Conference*. IEEE, 1–4.

[19] DPDK. 2018. Pktgen Traffic Generator Using DPDK. (aug 2018). http://dpdk.org/git/apps/pktgen-dpdk

[20] DPDK. 2019. DPDK burst replay tool. (aug 2019). https://github.com/FraudBuster/dpdk-burst-replay

[21] Rick Durrett. 2010. *Probability: Theory and Examples* (4 ed.). Cambridge University Press. https://doi.org/10.1017/CBO9780511779398

[22] Daniel Firestone. 2017. {VFP}: A Virtual Switch Platform for Host {SDN} in the Public Cloud. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 315–328.

[23] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. 2018. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 51–66.

[24] Brendan Gregg. 2017. Performance Superpowers with Enhanced BPF. USENIX Association, Santa Clara, CA.

[25] Sangjin Han. 2019. *System Design for Software Packet Processing*. Ph.D. Dissertation. University of California, Berkeley, Berkeley, CA.

[26] Oliver Hohlfeld, Johannes Krude, Jens Helge Reelfs, Jan Rüth, and Klaus Wehrle. 2019. Demystifying the Performance of XDP BPF. In *2019 IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 208–212.

[27] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. 2018. The EXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*. Association for Computing Machinery.

[28] Qun Huang, Patrick PC Lee, and Yungang Bao. 2018. Sketchlearn: relieving user burdens in approximate measurement with automated statistical inference. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 576–590.

[29] Nikita Ivkin, Ran Ben Basat, Zaoxing Liu, Gil Einziger, Roy Friedman, and Vladimir Braverman. 2020. I know what you did last summer: Network monitoring using interval queries. In *Abstracts of the 2020 SIGMETRICS/Performance Joint International Conference on Measurement and Modeling of Computer Systems*. 61–62.

[30] Rishabh Iyer, Katerina Argyraki, and George Candea. 2022. Performance Interfaces for Network Functions. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 567–584. https://www.usenix.org/conference/nsdi22/presentation/iyer

[31] Zachary H Jones. 2021. Performance Analysis of {XDP} Programs. *Large Installation System Administration Conference (LISA'21)* (2021).

[32] Snellman Juho. 2019. parallel-xxhash. https://github.com/jsnell/parallel-xxhash. (2019).

[33] Daniel Kelly, Frank Glavin, and Enda Barrett. 2020. Serverless Computing: Behind the Scenes of Major Platforms. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. IEEE, 304–312.

[34] Maurice George Kendall, Alan Stuart, and Keith Ord. 2010. *Kendall's Advanced Theory of Statistics* (6 ed.). Vol. 3. Oxford University Press.

[35] Praveen Kumar, Nandita Dukkipati, Nathan Lewis, Yi Cui, Yaogong Wang, Chonggang Li, Valas Valancius, Jake Adriaens, Steve Gribble, Nate Foster, and Amin Vahdat. 2019. PicNIC: Predictable Virtualized NIC. In *Proceedings of the ACM Special Interest Group on Data Communication*. Association for Computing Machinery.

[36] Joshua Levin and Theophilus A Benson. 2020. ViperProbe: Rethinking Microservice Observability with eBPF. In *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*. IEEE, 1–8.

[37] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. 2019. Nitrosketch: Robust and general sketch-based monitoring in software switches. In *Proceedings of the ACM Special Interest Group on Data Communication*. 334–350.

[38] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 101–114.

[39] Sebastiano Miano, Xiaoqi Chen, Ran Ben Basat, and Gianni Antichi. 2023. Fast In-kernel Traffic Sketching in eBPF - Artifact for CCR'23. (March 2023). https://doi.org/10.5281/zenodo.7701453

[40] Sebastiano Miano, Fulvio Risso, Mauricio Vásquez Bernal, Matteo Bertrone, and Yunsong Lu. 2021. A framework for eBPF-based network functions in an era of microservices. *IEEE Transactions on Network and Service Management* 18, 1 (2021), 133–151.

[41] Sebastiano Miano, Alireza Sanaee, Fulvio Risso, Gábor Rétvári, and Gianni Antichi. 2022. Domain Specific Run Time Optimization for Software Data Planes. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2022)*. Association for Computing Machinery, New York, NY, USA, 1148–1164. https://doi.org/10.1145/3503222.3507769

[42] Chris Misa, Walt O'Connor, Ramakrishnan Durairajan, Reza Rejaie, and Walter Willinger. 2022. Dynamic Scheduling of Approximate Telemetry Queries. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 701–717. https://www.usenix.org/conference/nsdi22/presentation/misa

[43] Quentin Monnet. 2018. Ever Deeper with BPF – An Update on Hardware Offload Support. https://www.netronome.com/blog/ever-deeper-bpf-update-hardware-offload-support/. (November 2018).

[44] Hun Namkung, Zaoxing Liu, Daehyeok Kim, Vyas Sekar, and Peter Steenkiste. 2022. SketchLib: Enabling Efficient Sketch-based Monitoring on Programmable Switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 743–759. https://www.usenix.org/conference/nsdi22/presentation/namkung

[45] Jonas Otten and Lorenz Bauer. 2020. Multidimensional fair-share rate limiting in BPF. https://www.linuxplumbersconf.org/event/7/contributions/677/. *Linux Plumbers Conference 2020* (September 2020).

[46] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. 2015. The design and implementation of open vswitch. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*. 117–130.

[47] Urban Reini. 2020. SMhasher: Hash function quality and speed test. https://github.com/rurban/smhasher. (2020).

[48] Gerald Rogers. 2014. Accelerating Network Intensive Workloads Using the DPDK netdev. http://openvswitch.org/support/ovscon2014/. (November 2014).

[49] Hugo Sadok, Zhipeng Zhao, Valerie Choung, Nirav Atre, Daniel S. Berger, James C. Hoe, Aurojit Panda, and Justine Sherry. 2021. We Need Kernel Interposition over the Network Dataplane. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. Association for Computing Machinery.

[50] Nikita Shirokov and Ranjeeth Dasineni. 2018. Open-sourcing Katran, a scalable network load balancer. https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer/. (May 2018).

[51] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. 2017. Heavy-Hitter Detection Entirely in the Data Plane *(SOSR '17)*. Association for Computing Machinery, New York, NY, USA, 164–176. https://doi.org/10.1145/3050220.3063772

[52] Alok Tiagi, Hariharan Ananthakrishnan, Ivan Porto Carrero, and Keerti Lakshminarayan. 2021. How Netflix uses eBPF flow logs at scale for network insight. https://netflixtechblog.com/how-netflix-uses-ebpf-flow-logs-at-scale-for-network-insight-e3ea997dca96. (June 2021).

[53] Linus Torvalds. 2003. Kernel floating-point. (March 2003). Retrieved June 6, 2022 from https://yarchive.net/comp/linux/kernel_fp.html

[54] William Tu, Joe Stringer, Yifeng Sun, and Yi-Hung Wei. 2018. Bringing the Power of eBPF to Open vSwitch. In *Linux Plumber Conference*.

[55] Juan Camilo Vega, Marco Antonio Merlini, and Paul Chow. 2020. FFShark: a 100G FPGA implementation of BPF filtering for Wireshark. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 47–55.

[56] VMware. 2020. VMware's per-CPU Pricing Model. https://www.vmware.com/company/news/updates/cpu-pricing-model-update-feb-2020.html. (March 2020).

[57] Siyao Zhao, Haoyu Gu, and Ali José Mashtizadeh. 2021. SKQ: Event Scheduling for Optimizing Tail Latency in a Traditional OS Kernel. In *Annual Technical Conference (ATC)*. USENIX Association.

A ARTIFACT APPENDIX

We aim at making the source code and all the script to run the different sketches and replicate the results available so that anyone can use and experiment them. In this respect, it is worth remembering that the performance characterization requires a careful prepared setup, including traffic generators and proper hardware devices (server machines, NICs).

A.1 How to access

Our artifacts are archived under Zenodo [39]. We also published the software and artifacts, with the corresponding setup instructions at the following Github repository (https://github.com/QMUL-EECS-Networks-Systems/ebpf-sketches).

Note: The GitHub repository is the preferred method to access our artifacts, since it provides the latest updates and fixes. The repository is linked with Zenodo so that every release is automatically archived.

Once downloaded the artifacts, please follow the instructions available under the `README.md`, which describes the installation process, and the experimental workflow to perform the different experiments.

A.2 Replicate paper's results.

Most of the experiments in our paper have been performed using a synthetic traffic trace with minimum sized packets, and the UNI1 data center trace from [10]. Under the `tests` folder of our artifacts, we provide all the scripts to run the tests and generate the results presented in the paper, including the instructions to configure the two traffic generators (i.e., `Pktgen-DPDK`[19] and `dpdk-burst-replay` [20]).