Fast In-kernel Traffic Sketching in eBPF

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

Xiaoli Chen  
Princeton University  
xiaoli@cs.princeton.edu

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

Gianni Antichi  
Queen Mary University London  
g.antichi@qmul.ac.uk

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 the need for 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 [1, 2, 17, 22, 32, 34]. A single physical server might host hundreds of VMs or thousands of containers, all connected to the network via a virtual switch [21, 44]. The virtual switch, therefore, plays a key role in ensuring the performance and security of the data center network [5]. 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 [8, 27, 36] 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 [44], Microsoft VFP [21]) 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 [26]. Furthermore, with DPDK, administrators must maintain two separate networking configurations, one for the kernel and one for DPDK, increasing their management burden [47, 55]. Finally, in some deployment environments, it may not be feasible to dedicate the CPU and memory resources DPDK requires [52], because of the high per-core pricing of current deployments [54].

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 [25, 26]. In fact, there are already several projects in industry [15] and academia [38] 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 [3, 40, 42], such as heavy hitters detection [8, 14, 49], per-flow frequency estimation [12, 16, 28], and counting distinct flows [4, 6, 37]. 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 the eBPF environment and its instructions set, the answers to those questions cannot be directly taken from best practices in user-space networking (§2).

We implemented NitroSketch [36] in eBPF, the state-of-the-art software-based solution that builds on top of Count Sketch [13] and Univmon [37]. 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);
Figure 1: Reducing the number of map_lookup calls by changing memory layout can significantly improve the performance of eBPF programs.

- Discuss how to close the performance gap between eBPF and user-space programs (§4);
- Share the code and all the benchmarks in open-source (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 map_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 [36]. 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.
- **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 dereferencing.
- **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 [36, 37] 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.

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

\(^1\)This number strictly depend on the specific sketching algorithm being used.

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

map is negligible, instead. Figure 1 shows the heavy performance penalty when the eBPF program needs to call the map_lookup function multiple times. However, when the eBPF program uses only one map_lookup call for multiple memory updates (case #3), its performance is 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 [36], 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 the 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 [44].

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.

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
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.

provide SIMD capabilities. In Figure 3, we benchmarked the performance of the vectorial version of xxHash and murmurhash hash functions when running as a normal user-space program with SIMD enabled, and we compared it with the eBPF. As we can see, in this case they performed the best in user-space, which is why they are chosen by prior works [36, 44]. 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 currently 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, 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 [30]. 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 memory. 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 [29, 39]. 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 [11]. 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 [46], VPP [7], and BESS [24].

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 [13] is used to produce unbiased estimation of flow sizes, while UnivMon [37] 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 higher 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 [36] 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 eBPF map. 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 [20, 35], which gives the probability that the first occurrence of success of a given event requires $k$ independent trials, each with success probability $p$. 
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 [9], which has an average packet size of 542 bytes (Figure 8). We used pktgen [18] with DPDK v20.11.0 to generate random 64B packets, the DPDK burst-replay [19] 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 [36], 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.

**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 our hardware (25.5 Mpps) 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...
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.

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 the Facebook’s open-source L4 eBPF/XDP load balancer Katran [48] 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 [43], 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 looku32 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 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 static linked 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 [43], a prototype using eBPF to track hierarchical heavy-hitter flows and rate-limit them. Meanwhile, Bertin also discussed an eBPF-based DDoS defense system [10] deployed at CloudFlare. Netflix built an eBPF-based network monitoring system [50] to produce flow logs. ViperProbe [35] 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 [23], there are not many prior work investigating the performance of eBPF programs themselves. Jones [30] 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 pseudorandom 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 [25]. For example, Netronome’s SmartNICs [41] allows offloading eBPF programs, and Vega et al. [53] 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 a 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.

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.
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).

Source code. The software of this paper is available at this URL: $ git@github.com/QMUL-EECS-Networks-Systems/ebpf-sketches

Once cloned the repository, please follow the instructions available under the README.md (https://github.com/QMUL-EECS-Networks-Systems/ebpf-sketches/blob/main/README.md), which describes the installation process, and the experimental workflow to perform the different experiments.

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 UNI data center trace from [9]. Under the tests (https://github.com/QMUL-EECS-Networks-Systems/ebpf-sketches/tree/main/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[18] and dpdk-burst-replay [19]) (https://github.com/QMUL-EECS-Networks-Systems/ebpf-sketches/tree/main/pkt-generator).