Supplementary material of eNetSTL

Bin Yang[1], Dian Shen[1*], Junxue Zhang[2*], Hanlin Yang[1], Lunqi Zhao[1], Beilun Wang[1], Guyue Liu[3], Kai Chen[2]
[1]Southeast University
[2]Hong Kong University of Science and Technology [3]Peking Univeristy

1 Core Components in NFs

Here we provide a detailed list of the core components we implemented in eBPF for these 35 representative works [1–15, 17–23, 25–37].

1. Cuckoo Hash [26]: We implemented the data structure query and insertion algorithms from section 2 of the paper using eBPF. We report the performance of the lookup algorithm. Specifically, during testing, we use a padded 16-byte 5-tuple as the key because the cuckoo hash is commonly employed as a connection table in network forwarding functions. In the kernel implementation, we optimized performance using hardware-assisted crc_hash and SIMD instruction-based full key comparison.
2. Cuckoo Switch [37] We implemented the extended Block Cuckoo Hash algorithm described in Section 2 of the paper as the key-value query operation using eBPF. The specific implementation incorporates appropriate modifications based on DPDK's cuckoo hash library [16]. In the kernel implementation, we optimized performance using hardware-assisted crc_hash, SIMD instruction-based full key comparison, and batch signature comparison.
3. d-ary Cuckoo [13]: We implemented the insertion and deletion algorithms for the d-ary Cuckoo Hash described in section 3 using eBPF. The testing methodology is similar to Cuckoo Hash. The kernel implementation utilizes SIMD parallel computing for optimizing multiple hash functions.
4. SILT [18]: We implemented the Partial-key Cuckoo hashing algorithm from Section 3.1 of the paper using eBPF. The specific implementation is referenced from DPDK's cuckoo hash lib, incorporating key ideas from the paper. The testing methodology is the same as for Cuckoo Hash. The kernel implementation utilizes hardware-assisted crc_hash and SIMD full key comparison for optimization.
5. NFD-HCS [21]: We attempted to implement the skiplist-based L1.cache (Key value query) from Section 3.1 of the paper using eBPF. However, due to eBPF's lack of support for non-contiguous memory, we were unable to implement it.
6. Bloom Filter [4]: We implemented the classic Bloom filter algorithm using eBPF as a membership test operation. In specific tests, we used five-tuples as keys to evaluate the performance of insertion and query operations for a single flow. The kernel implementation was optimized using SIMD-based parallel hash computation.
7. Summary Cache [12]: We implemented the memory membership test operation based on the Counter Bloom Filter, as described in Section V.C of the paper, using eBPF. The kernel version of the implementation was optimized using SIMD parallel hash calculations.
8. Cuckoo Filter [11]: We implemented the membership test operation based on the Cuckoo Filter, as proposed in Section 3 of the paper, using eBPF. Specifically, our implementation covers Algorithm 1, Algorithm 2, and Algorithm 3 from Section 3. We drew inspiration from the Cuckoo Filter implementation in DPDK [16]. The kernel version of our implementation leverages SIMD-based parallel comparisons and hardware-assisted crc_hash for optimization.
9. d-left BF [5]: We implemented the d-left Bloom Filter based on d-left hashing as proposed in Section 3 of the paper using eBPF. The kernel implementation was optimized using hardware-assisted crc_hash and SIMD-based parallel comparison.
10. Rank-Indexed [14]: We implemented the Bloom Filter construction algorithm, which is based on Rand-index hashing as proposed in Section 3 of the paper, for the membership test operation using eBPF. Our implementation encompasses the data structure illustrated in Figure 3 of the paper. The kernel implementation optimizes performance through the utilization of hardware POPCNT bitwise operations.
11. Blocked BF [27]: We implemented the Block Bloom Filter algorithm described in Section 3 of the paper using eBPF. The kernel implementation utilizes SIMD

*Corresponding authors.

EuroSys '25, March 30–April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1196-1/25/03
https://doi.org/10.1145/3689031.3696094

parallel comparisons and parallel hash calculations for optimization.

12. VBF [17]: We ported the multi-set membership test operation based on a vector of Bloom filters from DPDK to eBPF. The kernel implementation optimizes performance by utilizing hardware FFS instructions and hardware-assisted crc_hash.
13. Hypercut [31]: We implemented the search algorithm of Hypercut mentioned in Section 5.2 in the data plane using eBPF. According to the characteristics of this algorithm, the decision tree is constructed in user space and then directly written into the BPF map.
14. Efficut [34]: We implemented the search algorithm of Efficut decision tree mentioned in Section 3 of the paper using eBPF. We utilized BPF_MAP_TYPE_ARRAY-_OF_MAPS to store multiple decision trees separated by the algorithm.
15. TSS [32]: We implemented the Tuple Space Search algorithm from Section 5 of the paper using eBPF. Similar to OvS, we use Cuckoo Hash as the underlying hash table. The kernel implementation employs SIMD-based parallel comparisons and hardware-assisted crc_hash for optimization.
16. Maglev [9]: We implemented the Consistent Hashing algorithm from Section 3.4 of this paper using eBPF. Since this algorithm is based on table query, the calculation of table entries can be done in user space, and the eBPF data path only needs to implement table lookup operations. This can be achieved using BPF_MAP_TYPE_ARRAY or BPF_MAP_TYPE_PERCPU_ARRAY easily just as the implementation in Katran [24].
17. Beamer [25]: We implemented the stable hashing algorithm from Section 4.1 of the paper using eBPF. Similar to Maglev, stable hash is also a table-based load balancing algorithm, making it easily implementable with eBPF. Additionally, since stable hash adds an extra direction mapping, it can be implemented using the BPF_MAP_TYPE_ARRAY_OF_MAP.
18. EFD [7]: We ported DPDK's EFD, a load balancing library based on perfect hash, to eBPF. In the data path, EFD involves only two hash calculations. The selection and update of the perfect hash functions for the table are done in user space. We tested the lookup performance of the eBPF implementation. For the kernel implementation, we optimized the performance using hardware-assisted crc_hash for optimization.
19. SpaceSaving [22]: We attempted to implement the Stream Summary data structure for counting proposed in Section 3.1 of this paper using eBPF, Stream Summary involves a significant number of pointer operations and non-contiguous memory, making it impractical for implementation in eBPF.
20. CSS [2]: We implemented the Compact Space Saving (CSS) data structure from section III.B of the paper using eBPF to achieve counting. We utilized Blocked Cuckoo Hash as the static hash table for CSS. CSS was deployed in the data path, and for each incoming packet, we inserted it into CSS to test insertion performance. In the kernel implementation, we employed SIMD parallel comparisons and hardware-assisted crc_hash for optimization.
21. HSS [23]: We implemented the Hierarchical Heavy Hitters based on the Space Saving algorithm from Section 3 of the paper using eBPF. Specifically, for the Space Saving algorithm, we adopted the pointless CSS data structure instead of Stream Summary. In the kernel implementation, we optimized the CSS hash table indexing using SIMD.
22. RHSS [3]: We implemented Algorithm 1, Randomized HHH, from the paper using eBPF for counting. Additionally, we tested the insertion performance of RHHH. In the eBPF implementation, we utilized `bpf_prandom_get_u32` for generating random numbers. Meanwhile, in the kernel implementation, we optimized it using a GEO random number pool.
23. Tiny Table [8]: We implemented the Tiny Table data structure described in Section 3.1 of the paper, specifically the structures depicted in Figures 1, 2, and 3, using eBPF. We tested its insertion performance in the eBPF data path. In the kernel implementation, we optimized bit searches using FFS hardware instructions and improved bucket lookup with SIMD instructions.
24. Memento [1]: We implemented the Memento algorithm proposed in Section 4.1 of the paper using pure eBPF to support counting with a sliding window. In the eBPF implementation, we used `bpf_prandom_-get_u32` for generating random numbers. In the kernel implementation, we optimized it using a GEO random number pool. Additionally, we utilized SIMD parallel comparisons to optimize the underlying CSS data structure used by Memento.
25. HeavyKipper [36]: We implemented the HeavyKeeper data structure from Section III.B of the paper using pure eBPF. In the data plane, we tested its insertion performance using the five-tuple as the key. In the kernel implementation, we optimized the overall performance of HeavyKeeper using hardware-assist crc_hash and SIMD-based parallel comparisons.
26. Count-min Sketch [6]: We implemented the Count-Min Sketch algorithm from Section 3 of the original paper using eBPF. In the eBPF data path, we tested the algorithm's insertion performance using the five-tuple of packet data as keys. In the kernel implementation, we optimized it using SIMD parallel hash computations.

27. UnivMon Sketch [20]: We implemented Algorithm 1, the UnivMon Online Sketching Step, from Section 4.2 of the paper using eBPF. We used Count-Min as the underlying L2-Heavy-Hitter Sketch. In the eBPF implementation, we used a static hash table and a binary heap to maintain the Top-K. In the kernel implementation, we optimized performance using SIMD parallel hash computations. When K is set to a small value, we used SIMD parallel comparisons and Reduce Min to implement the heap; otherwise, we used the same approach as in eBPF.
28. Sketch Visor [15]: We implemented the sketch visor's fast path and slow path in Section 3 of the paper using eBPF. Specifically, we utilized the Count-min sketch as the slow path. Additionally, we implemented Algorithm 1 in Section 4.2, i.e., the fast path of Sketch Visor, based on the hash table provided by eBPF. In the kernel implementation, we optimized the slow path by employing SIMD parallel hash calculations. The fast path was optimized by using a static hash table and employing SIMD parallel comparisons.
29. Elastic Sketch [35]: We implemented the basic algorithm of Elastic Sketch in Section 3.1 of the paper using eBPF, specifically the heavy part and light part as depicted in Figure 1. We adopted the original paper's approach of using Count Min as the light part. In the kernel implementation, we utilized SIMD parallel calculations for the light part. Following the methodology outlined in Section 4.3 of the paper, we employed static arrays along with SIMD parallel comparisons to implement the heavy part.
30. Nitro Sketch [19]: We implemented the core algorithm Nitro Sketch from Section 4.1 of the paper using eBPF. Specifically, in eBPF, we employed `bpf_prandom_get_u32` to determine whether to update a particular row, as illustrated in Figure 4 in the origin paper. In the kernel implementation, we followed the approach outlined in Figure 7(b) of the paper, utilizing a GEO random number pool and incorporating hardware-assisted crc hash to accelerate hash calculations.
31. Eiffel [29]: We attempted to implement the priority queue algorithm cFFS from Section 3.1 of the paper using eBPF. Although the current eBPF does not support packet queuing, we can achieve this by leveraging existing patches that have not yet been merged into the kernel mainline. In the eBPF implementation, we referred to the implementation of the software FFS instruction in Linux, using several bitwise AND operations to achieve the functionality of FFS. In the kernel implementation, we optimized the performance of cFFS by using hardware-based FFS instructions.
32. PCQ [30]: We implemented the core data structure, a calendar queue, used for packet queuing in the Section 3 of the paper using eBPF. In the eBPF implementation, we utilized eBPF linked lists and BPF arrays to implement the calendar queue.
33. Carousel [28]: We implemented the core queuing algorithm, the time wheel described in Section 4 of the paper using eBPF. For the implementation of the time wheel, we referred to the time wheel implementation in Linux kernel. In the eBPF implementation, we used BPF linked lists and BPF arrays. Additionally, we utilized `bpf_ktime_get_ns` to obtain timestamps.
34. Non-cascade Tw [33]: We ported the non-cascading time wheel algorithm from Linux kernel to eBPF. We utilized eBPF linked lists and software-simulated bitwise operations to implement the non-cascading time wheel. In the kernel implementation, we optimized it using hardware-based FFS instructions.
35. FQ/pacing [10]: We attempted to implement the packet queuing algorithm based on a red-black tree in Linux using eBPF. However, due to the lack of support for non-contiguous memory in eBPF, it is not feasible to implement a red-black tree within eBPF.

References

[1] Ran Ben Basat, Gil Einziger, Isaac Keslassy, Ariel Orda, Shay Vargaftik, and Erez Waisbard. 2018. Memento: Making Sliding Windows Efficient for Heavy Hitters. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '18)*. Association for Computing Machinery, New York, NY, USA, 254–266. https://doi.org/10.1145/3281411.3281427

[2] Ran Ben-Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. 2016. Heavy hitters in streams and sliding windows. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*. 1–9. https://doi.org/10.1109/INFOCOM.2016.7524364

[3] 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

[4] Burton H. Bloom. 1970. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (jul 1970), 422–426. https://doi.org/10.1145/362686.362692

[5] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. 2006. An improved construction for counting bloom filters. In *Algorithms–ESA 2006: 14th Annual European Symposium, Zurich, Switzerland, September 11-13, 2006. Proceedings 14*. Springer, 684–695.

[6] Graham Cormode and S. Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75. https://doi.org/10.1016/j.jalgor.2003.12.001

[7] DPDK. Accessed Jan. 2024. Elastic Flow Distributor Library. https://doc.dpdk.org/guides/prog_guide/efd_lib.html. (Accessed Jan. 2024).

[8] Gil Einziger and Roy Friedman. 2016. Counting with tinytable: Every bit counts!. In *Proceedings of the 17th International Conference on Distributed Computing and Networking*. 1–10.

[9] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. 2016. Maglev: A fast and reliable software network load balancer. In *13th {USENIX} Symposium on*

Networked Systems Design and Implementation ({NSDI} 16). 523–535.

[10] Eric Dumazet. Accessed Jan. 2024. pkt_sched: fq: Fair Queue packet scheduler. https://lwn.net/Articles/564825/. (Accessed Jan. 2024).

[11] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. 2014. Cuckoo Filter: Practically Better Than Bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies (CoNEXT '14)*. Association for Computing Machinery, New York, NY, USA, 75–88. https://doi.org/10.1145/2674005.2674994

[12] Li Fan, Pei Cao, J. Almeida, and A.Z. Broder. 2000. Summary cache: a scalable wide-area Web cache sharing protocol. *IEEE/ACM Transactions on Networking* 8, 3 (2000), 281–293. https://doi.org/10.1109/90.851975

[13] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul Spirakis. 2005. Space efficient hash tables with worst case constant access time. *Theory of Computing Systems* 38, 2 (2005), 229–248.

[14] Nan Hua, Haiquan Zhao, Bill Lin, and Jun Xu. 2008. Rank-indexed hashing: A compact construction of Bloom filters and variants. In *2008 IEEE International Conference on Network Protocols*. 73–82. https://doi.org/10.1109/ICNP.2008.4697026

[15] Qun Huang, Xin Jin, Patrick P. C. Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. 2017. SketchVisor: Robust Network Measurement for Software Packet Processing. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 113–126. https://doi.org/10.1145/3098822.3098831

[16] Intel. Accessed Jan. 2024. Hash Library. https://doc.dpdk.org/guides/prog_guide/hash_lib.html. (Accessed Jan. 2024).

[17] Intel. Accessed Jan. 2024. Membership Library. https://doc.dpdk.org/guides/prog_guide/member_lib.html. (Accessed Jan. 2024).

[18] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. 2011. SILT: A Memory-Efficient, High-Performance Key-Value Store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/2043556.2043558

[19] 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 (SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 334–350. https://doi.org/10.1145/3341302.3342076

[20] 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 (SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 101–114. https://doi.org/10.1145/2934872.2934906

[21] Rodrigo B. Mansilha, Lorenzo Saino, Marinho P. Barcellos, Massimo Gallo, Emilio Leonardi, Diego Perino, and Dario Rossi. 2015. Hierarchical Content Stores in High-Speed ICN Routers: Emulation and Prototype Implementation. In *Proceedings of the 2nd ACM Conference on Information-Centric Networking (ACM-ICN '15)*. Association for Computing Machinery, New York, NY, USA, 59–68. https://doi.org/10.1145/2810156.2810159

[22] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2005. Efficient computation of frequent and top-k elements in data streams. In *Database Theory-ICDT 2005: 10th International Conference, Edinburgh, UK, January 5-7, 2005. Proceedings 10*. Springer, 398–412.

[23] Michael Mitzenmacher, Thomas Steinke, and Justin Thaler. 2012. Hierarchical heavy hitters with the space saving algorithm. In *2012 Proceedings of the Fourteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 160–174.

[24] Nikita Shirokov, Ranjeeth Dasineni. Accessed May. 2024. 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/. (Accessed May. 2024).

[25] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. 2018. Stateless datacenter load-balancing with beamer. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 125–139.

[26] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *Journal of Algorithms* 51, 2 (2004), 122–144. https://doi.org/10.1016/j.jalgor.2003.12.002

[27] Felix Putze, Peter Sanders, and Johannes Singler. 2007. Cache-, hash- and space-efficient bloom filters. In *Experimental Algorithms: 6th International Workshop, WEA 2007, Rome, Italy, June 6-8, 2007. Proceedings 6*. Springer, 108–121.

[28] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. 2017. Carousel: Scalable Traffic Shaping at End Hosts. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 404–417. https://doi.org/10.1145/3098822.3098852

[29] Ahmed Saeed, Yimeng Zhao, Nandita Dukkipati, Ellen Zegura, Mostafa Ammar, Khaled Harras, and Amin Vahdat. 2019. Eiffel: Efficient and Flexible Software Packet Scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 17–32. https://www.usenix.org/conference/nsdi19/presentation/saeed

[30] Naveen Kr. Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. 2020. Programmable Calendar Queues for High-speed Packet Scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 685–699. https://www.usenix.org/conference/nsdi20/presentation/sharma

[31] Sumeet Singh, Florin Baboescu, George Varghese, and Jia Wang. 2003. Packet classification using multidimensional cutting. In *Proceedings of the ACM SIGCOMM 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 25-29, 2003, Karlsruhe, Germany*, Anja Feldmann, Martina Zitterbart, Jon Crowcroft, and David Wetherall (Eds.). ACM, 213–224. https://doi.org/10.1145/863955.863980

[32] V. Srinivasan, S. Suri, and G. Varghese. 1999. Packet Classification Using Tuple Space Search. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '99)*. Association for Computing Machinery, New York, NY, USA, 135–146. https://doi.org/10.1145/316188.316216

[33] Thomas Gleixner. Accessed Jan. 2024. timer: Refactor the timer wheel. https://lore.kernel.org/lkml/20160617121134.417319325@linutronix.de/. (Accessed Jan. 2024).

[34] Balajee Vamanan, Gwendolyn Voskuilen, and T. N. Vijaykumar. 2010. EffiCuts: optimizing packet classification for memory and throughput. In *Proceedings of the ACM SIGCOMM 2010 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, New Delhi, India, August 30 -September 3, 2010*, Shivkumar Kalyanaraman, Venkata N. Padmanabhan, K. K. Ramakrishnan, Rajeev Shorey, and Geoffrey M. Voelker (Eds.). ACM, 207–218. https://doi.org/10.1145/1851182.1851208

[35] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. 2018. Elastic Sketch: Adaptive and Fast Network-Wide Measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 561–575. https://doi.org/10.1145/3230543.3230544

[36] Tong Yang, Haowei Zhang, Jinyang Li, Junzhi Gong, Steve Uhlig, Shigang Chen, and Xiaoming Li. 2019. HeavyKeeper: An Accurate Algorithm for Finding Top- *k* Elephant Flows. *IEEE/ACM Transactions on Networking* 27, 5 (2019), 1845–1858. https://doi.org/10.1109/TNET.2019.2933868

[37] Dong Zhou, Bin Fan, Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2013. Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT '13)*. Association for Computing Machinery, New York, NY, USA, 97–108. https://doi.org/10.1145/2535372.2535379