# Understanding Performance of eBPF Maps

Chang Liu
chang-liu22@mails.tsinghua.edu.cn
Tsinghua University
Beijing, China

Byungchul Tak
bctak@knu.ac.kr
Kyungpook National University
Daegu, Republic of Korea

Long Wang
longwang@tsinghua.edu.cn
Tsinghua University &
Zhongguancun Laboratory
Beijing, China

## ABSTRACT

The Linux community has witnessed the rapid development of eBPF technology that allows users to load custom programs into the Linux kernel to extend its capabilities. A key feature that makes eBPF powerful is eBPF maps, which provide data storage and communication capabilities for eBPF programs. However, despite being widely used in eBPF programs, the performance of eBPF maps has received little attention. To understand the performance characteristics of eBPF maps, we conduct a comprehensive benchmark on them. The benchmark results demonstrate the access overhead of different types of eBPF maps and reveal the impact of various factors on the access overhead. By analyzing the benchmark results, we derive some implications for eBPF users to use eBPF maps more efficiently.

## CCS CONCEPTS

• **General and reference** → **Measurement**; **Performance**.

## KEYWORDS

eBPF, performance benchmark, eBPF map

**ACM Reference Format:**
Chang Liu, Byungchul Tak, and Long Wang. 2024. Understanding Performance of eBPF Maps. In *Workshop on eBPF and Kernel Extensions (eBPF '24), August 4–8, 2024, Sydney, NSW, Australia.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3672197.3673430

## 1 INTRODUCTION

The emerging eBPF technology has seen widespread adoption across many use cases. eBPF allows users to run custom programs in Linux kernel at runtime to extend its capabilities safely, unlike kernel modules whose code flaws may crash the kernel. Nowadays, there exist a large number of eBPF-based projects that have been widely deployed in production environments and provide diverse functionalities including tracing [1, 2], auditing [5, 8], container networking [3], load balancing [6] and DDoS protection [7]. Researchers also leverage eBPF to implement network [16–18, 24], storage [11, 14, 20, 23], security [13, 19] and virtualization [10, 21] applications within Linux kernel.

Several new features of eBPF make it attractive for various kernel tracing tasks. eBPF allows users to write custom programs and attach them to a wide range of pre-defined in-kernel hooks. Before being attached to the kernel, eBPF programs undergo verification by the eBPF verifier. The verifier rejects programs containing operations that may affect the kernel's stability, including out-of-bound memory access and unbounded loop, thus guaranteeing the safety of eBPF programs. For stateful processing, eBPF provides eBPF maps that implement common data structures and allow access from both eBPF and userspace programs.

The eBPF technology continues to evolve rapidly. Kernel developers continuously submit kernel patches that implement new eBPF features, including new hooks, new map types, and new helper functions. Despite eBPF's functionality being a focal point, its performance is often overlooked. Most eBPF kernel patches, eBPF kernel documentation, and eBPF tutorials do not contain information about eBPF performance. With a limited understanding of eBPF performances, developers face the following questions: *i) How much performance impact will there be for my eBPF applications due to the eBPF mechanism? ii) What is the most efficient way of using eBPF? E.g., which eBPF map among several options is the most efficient for my workload? iii) How can I further optimize my eBPF programs to achieve higher performance and scalability?*

Central to eBPF performance evaluation is the assessment of eBPF maps. These maps serve as the primary means of data storage and communication within eBPF programs, making their performance characteristics crucial in determining the overall efficiency of eBPF applications. However, it is not straightforward to deduce the critical factors that affect the performance of eBPF programs. Some potential factors cannot be known without benchmarking; for example, we find


*eBPF '24, August 4–8, 2024, Sydney, NSW, Australia*

ACM ISBN 979-8-4007-0712-4/24/08…$15.00
https://doi.org/10.1145/3672197.3673430

that per-cpu map suffers from memory access overhead amplification when inserting new keys. Some intuitions turn out to be incorrect. For example, writing 4 bytes to a map is expected to be faster than writing 256 bytes, but our test results show that their overheads are nearly the same.

In this work, we conducted a comprehensive benchmarking study to understand the performance of eBPF maps. We aimed to learn the impact of various factors on their access overheads. Our benchmark results provide both theoretical results derived from running eBPF programs in an artificial context and practical results by triggering eBPF programs with synthetic workloads. Furthermore, by analyzing the benchmark results, we derived some implications for eBPF users to help make their eBPF applications more efficient.

Our contribution can be summarized as follows: First, we present comprehensive benchmark results on eBPF maps to reveal the performance characteristics and overheads. We find that memory footprint and cache hotness are key factors affecting the overhead of eBPF maps. When the memory footprint is small, the overhead of accessing eBPF maps under a cold cache is an order of magnitude higher than under a hot cache. Second, we derive some implications from the benchmark results in using eBPF maps more efficiently. For example, we find that eBPF has the "volume discount" feature, which means attaching an eBPF program to more hooks reduces its amortized execution time cost. A system call tracing example shows that when attached to three system calls, the execution time of the eBPF program is 35%-84% of when attached to a single system call.

## 2 THE BENCHMARK SYSTEM

In this section, we provide details on how we conducted our benchmark on eBPF maps. We first explain the experimental setup, followed by the design of our automated benchmark system. We outline several caveats of our benchmark. First, eBPF is now available on platforms running on different operating systems, i.e., Linux and Windows [4], as well as different hardware, including general-purpose CPUs and SmartNICs [12, 15]. In Linux, in addition to attaching to kernel hooks, eBPF programs can also run in userspace with support of userspace runtimes [9, 22]. Among these platforms and contexts, our benchmark focuses on the eBPF in Linux kernel on x86-64 CPUs with JIT enabled. As JIT is enabled by default, our study focuses on the performance in this most popularly used deployment for eBPF, and do not consider the performance of eBPF native code. Second, our benchmark system can be applied onto different Linux kernel versions, though the measured performance results may be different on them because the eBPF implementation/features evolve rapidly as Linux kernel updates. (Our benchmark provides consistent results on a specific kernel version, and we did the experiments in Linux 6.0.1). Third,

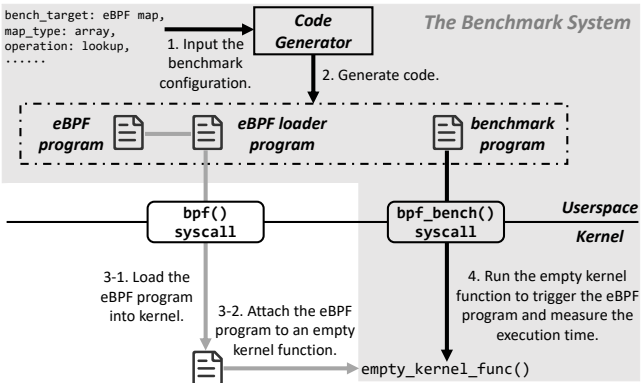


**Figure 1: System Overview.**

some benchmark results, e.g., the execution time, are closely related to the hardware specifications of our system, e.g., CPU frequency, memory frequency, and cache size. These results can not be generalized to other hardware settings. However, conclusions derived from these results are general.

### 2.1 Testbed Setup

Our benchmark server has two Intel Xeon Silver 4210R CPUs with 2.40 GHz base frequency. Each CPU has 10 physical cores. The sizes of L1i, L1d, L2, and L3 cache are 32 KB, 32 KB, 1 MB, and 13.75 MB respectively. The server also has 128 GB DDR4 memory with 3200 MHz frequency. We disable dynamic P-states and C-states of the server to keep it running at a fixed frequency, i.e., the base frequency, which ensures the accuracy and stability of the benchmark results.

### 2.2 System Overview

Figure 1 shows the overview of our benchmark system. The benchmark system consists of two parts: the *code generator* and the `bpf_bench()` system call.

The code generator reads the user-specified benchmark configuration and generates *i)* an eBPF program to be tested, *ii)* an eBPF loader program, and *iii)* a benchmark program. The configuration includes the attributes of the tested eBPF map, e.g., map type, map size, value size and the operation on the map (*lookup or update*). An example of the configuration is shown on the top left corner of Figure 2. The other parts of Figure 2 show the generated eBPF program, i.e. the *map lookup benchmark program* for the *lookup* operation in the configuration, or the *map update benchmark program* for the *update* operation, together with the definitions of maps required by the eBPF program.

The eBPF loader program loads this eBPF program into the kernel and attaches it to an empty kernel function that we implement as its hook point (the *kprobe/empty_kernel_func* bound to the eBPF programs in Figure 2). The benchmark program in the user space then invokes the `bpf_bench()` system call once to do the benchmarking.

**Table 1: Setups for benchmarking eBPF maps. The last column shows the index of the sub-figure in Figure 3 corresponding to the benchmark result of each setup.**

| Map Type | Map Attributes⋆ | Operation | Result |
|---|---|---|---|
| array | map size | lookup | 3-a |
| | | update | 3-b |
| per-cpu array | | lookup | 3-c |
| | | update | 3-d |
| hash | key size<br>map size | lookup | 3-e<br>3-f |
| | key size<br>map size | update | 3-g<br>3-h |
| per-cpu hash | map/key size | lookup | 3-i |
| | | update | 3-j |
| ring buffer | map size | output<br>submit | 3-k |
| perf buffer | map size | output | 3-l |
| queue | map size | push | 3-m |
| | | pop/peak | 3-n |
| stack | | push | 3-o |
| | | pop/peak | 3-p |

⋆ The **value size** map attribute is included in each setup, so we omitted it from the second column to save space.

The measurement of the execution time of the attached eBPF program happens within the kernel. Thus, we implement a system call, `bpf_bench()`, to provide an interface to userspace. A single invocation of the `bpf_bench()` system call runs a loop that calls the empty kernel function continuously, and each call of the function triggers one execution of the eBPF program. We trigger the eBPF program many times within a single system call in order to access many eBPF entries, and hence, accurately measure the one-time eBPF program performance by averaging the multiple eBPF program executions. To benchmark the eBPF program under hot cache, `bpf_bench()` calls the empty kernel function continuously. For cold cache, `bpf_bench()` flushes the CPU caches explicitly by executing the `wbinvd` x86 instruction after each execution of the eBPF program, within the loop.

The `bpf_bench()` system call also supports benchmarking the overhead of concurrent execution of eBPF programs. Users only need to invoke `bpf_bench()` multiple times on different CPU cores. It synchronizes the parallel execution threads and ensures they execute the benchmarked eBPF program concurrently. All these features of the `bpf_bench()` system call are encapsulated by the code generator and exposed to users through the benchmark configuration interface. According to the user's intent in the benchmark configuration, the code generator generates a benchmark program containing the call to `bpf_bench()` along with the eBPF and loader program. Running the benchmark program starts the benchmark and yields the result.

The Linux kernel provides a native method to benchmark eBPF programs in the `bpf()` system call. Users can instruct the kernel to run a loaded eBPF program multiple times and measure the total execution time. However, this method cannot benchmark the impact of cache hotness on the overhead of eBPF programs because the eBPF programs always run with a hot cache. Additionally, this method also cannot measure the overhead of eBPF programs when running concurrently. These motivated us to implement `bpf_bench()` system call to benchmark eBPF overhead.

Besides measuring the eBPF programs and collecting the results by invoking the empty kernel function, we also conducted benchmarks where eBPF programs were attached to real kernel hooks and triggered by synthetic workloads. They are given in the following section.

## 3 BENCHMARKING EBPF MAPS

In this section, we describe the details of our benchmark on eBPF maps. We first discuss the factors we benchmarked that may affect the overhead of eBPF maps. Then, we present the benchmark results and the analysis. Note that the hash maps used in our experiments are pre-allocated ones.

### 3.1 Benchmark Setup

Different map types, operations, and map attributes all affect the overhead of accessing eBPF maps. Our benchmark covers these factors, and we list all combinations of these factors in Table 1. For each setup in Table 1, we test the map access overhead under both hot CPU caches and cold CPU caches. The method for controlling the hotness of CPU caches has been discussed in Section 2.2. In addition to the setups in Table 1, we also measured the overhead of executing an empty BPF program (null eBPF program lines in Figure 3). The result is used as a baseline that represents the overhead introduced by the eBPF kernel infrastructure to execute an eBPF program, i.e., the overhead of the trampoline code. By subtracting the empty BPF program overhead, we can obtain the pure overhead of eBPF map access operations.

Some eBPF applications use maps with large value sizes to store long data, e.g., network packets. To cover such cases, our map benchmark included large value sizes, e.g., 4KB. However, eBPF provides only a 512-byte stack, disallowing us to define a variable larger than 512 bytes as a value to be written into an eBPF map. To get around this limitation, we allocated the memory of the value in another map and accessed the map to obtain the pointer to the value's memory region before using it. To get consistent results, we stored values (as well as keys for hash map) of any size in an *array map*, even if the value could be allocated on the eBPF stack. This design required us to understand the additional overhead introduced by retrieving a value from an array map compared to directly using a value allocated on the stack.

The overhead of reading an element from an array map is shown in Figure 3-a. The overhead of retrieving an element from an array map is constant and is not affected by the

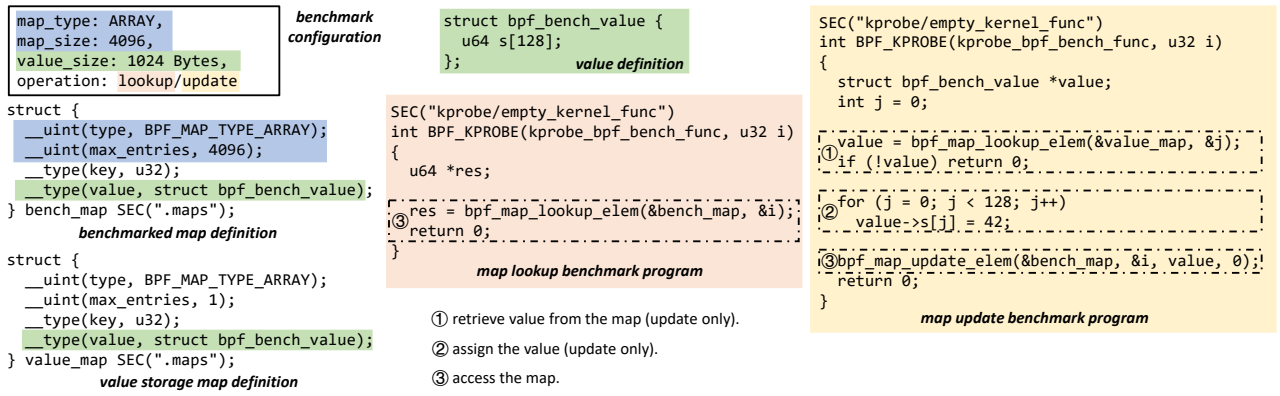


**Figure 2: An example of generated eBPF map benchmark program. Colors indicate the mapping between configuration items and the corresponding generated code snippets.**

map size or the value size irrespective of cache hotness. This is because an array map stores elements in a continuous memory region, which allows the eBPF JIT-compiler to inline the helper function call to get an element with a specific index into a few arithmetic instructions that calculate a memory address of the element. When the cache is hot, the overhead of executing these arithmetic instructions is negligible (at the bottom of Figure 3-a, the map read result lines and the null eBPF program overhead line overlap). When the cache is cold, these arithmetic instructions introduce an overhead of ~100 ns (the difference between the map read result lines and the null eBPF program line at the top of Figure 3-a), which is the overhead of loading instructions into the I-cache. These represent the overhead of using an array map to store the map values to support large value sizes.

Figure 2 shows an example of generating eBPF programs to benchmark array map lookup and update. For *read operations* on different map types, such as *lookup* on the array and *pop* on stack/queue operations, we measured the overhead of the corresponding eBPF helper function (③ in Figure 2). For *write operations*, i.e., *update* on array, *output* on ring/perf buffer, and *push* on stack/queue, we also measured the overhead of retrieving the value from the map (①) and assigning the value (②), which are prerequisites for map writing. The *bpf_bench()* system call passes an incrementing value (the *i* parameter in Figure 2) to the eBPF program, which is used to generate accesses to different map elements.

## 3.2 Benchmark Results of eBPF Maps

Figure 3 shows the access overhead of different eBPF map types. We first discuss the results for each type of map and then provide a summary on the eBPF map overhead. As mentioned in Section 2.2, one advantage of our benchmark system over the Linux native eBPF benchmark is to support concurrent scenarios. Unfortunately, due to the page constraints, we omit the results for concurrent scenarios.

*3.2.1 Array and Per-cpu Array.* The overhead of accessing 1024 entries of the array and per-cpu array maps is shown in Figure 3-a to Figure 3-d. We discussed array lookup in Section 3.1. For per-CPU array, its lookup overhead is also constant but is slightly higher than that of the array. This is because accessing a per-CPU array requires obtaining the current CPU ID, which also prevents the corresponding helper function from being inlined like the array lookup case. For updates on array and per-cpu array maps, the overhead increases with the value size which is natural because larger value sizes mean higher memory copy overhead. Most of the remaining map types exhibit a similar trend of the write overhead that varies with the value size. Thus, we omit further explanations for the analogous behaviors.

Due to its low lookup overhead, array maps can be used as a means to quickly allocate memory in eBPF programs, similar to how user-space processes allocate memory on the heap. Additionally, users can pre-stored data structures required by specific algorithms in an array map and access the data structure efficiently in eBPF programs to implement these algorithms. For example, users can store the state machine used by string matching algorithms into an array map and access it in eBPF programs for string matching, which makes it possible to offload functions like deep packet inspection and application layer protocol parsing with eBPF.

*3.2.2 Hash and Per-cpu Hash.* Figure 3-e to Figure 3-h show the overhead of accessing 1024 entries of the hash map. From the result we can draw the following observations: *i)* The lookup/update overhead of the hash map increases with the key size because calculating the hash value of larger keys incurs higher overhead and *ii)* the lookup/update overhead of the hash map decreases with the increase in map size. This is because the hash map is implemented by hash buckets, and the number of buckets is determined by rounding up the map size to the next power of two. A larger map size implies a greater number of buckets and fewer hash collisions, thus reducing the access overhead of the hash map.

Figure 3-i and Figure 3-j show the overhead of accessing the per-cpu hash map, which shows a similar impact of key

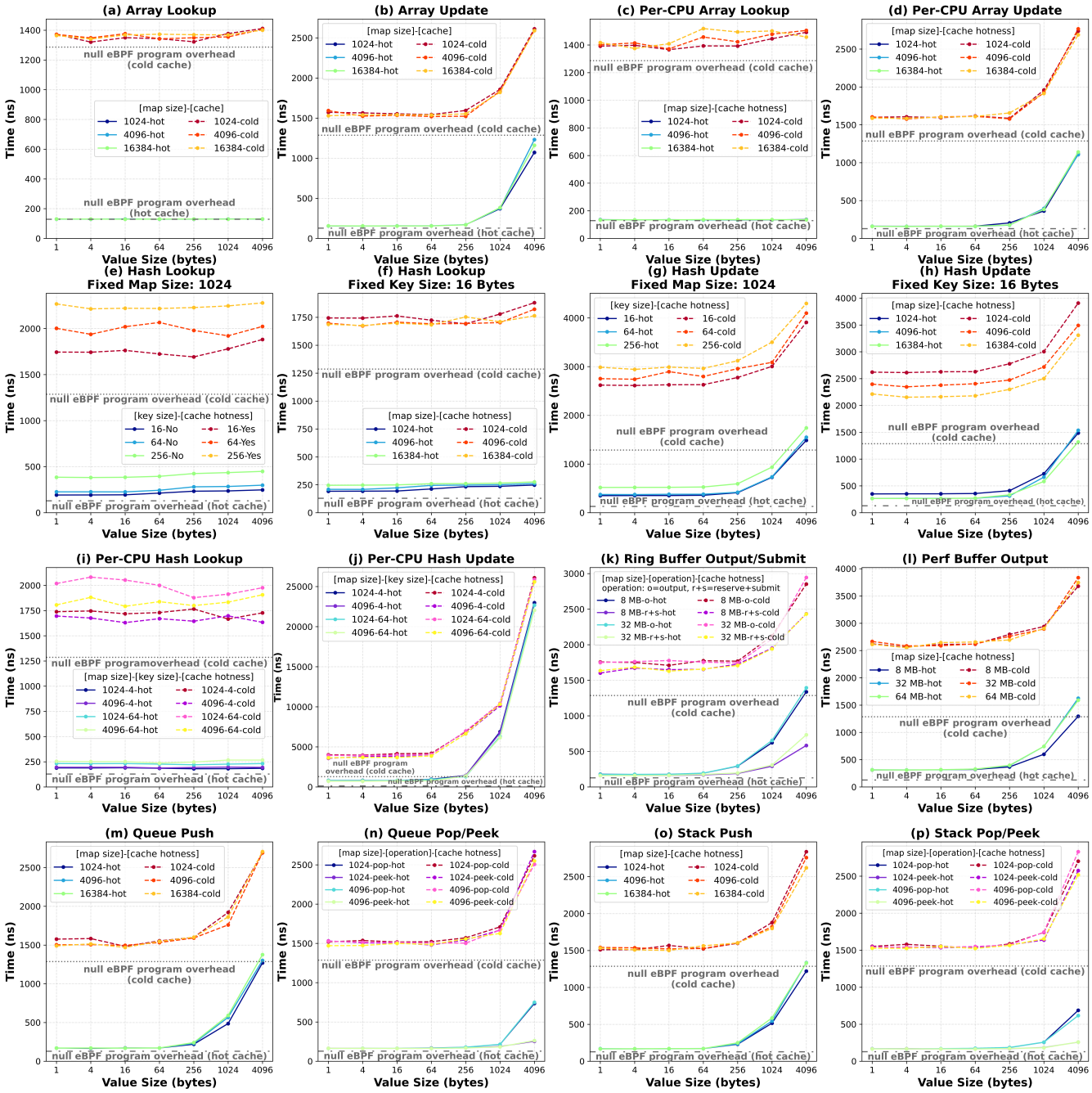


**Figure 3: Benchmark results of eBPF maps. As value ranges in the figures are quite different, for cosmetic purpose the scales of the axes in these figures are also not the same. In each experiment, we accessed 1024 different entries, i.e. accessed the map 1024 times, and calculated the average value. We conducted five experiments for each setup.**

size and map size to the hash map. The update overhead of the hash map increases rapidly with the value size. By scrutinizing the source code, we find that this is because, for each key, the per-cpu hash map uses a memory region to store values corresponding to each CPU core. When inserting a key-value pair to a per-cpu hash map, if the key does not exist in the map, the values corresponding to all other CPUs are initialized to 0, which means the memory access overhead is amplified besides updating the value corresponding to the current CPU. This characteristic of the per-CPU hash map implies that it should not be used in situations where new keys are frequently inserted and the value size is large.

**Table 2: Results of the system call tracing experiment.**

| Traced Syscall(s) | Run Count | Total Time (ns) | Avg. Time (ns) |
|---|---|---|---|
| *pread64()* | 6784 | 2032298 | 299.6 |
| *read()* | 84782 | 10665819 | 125.8 |
| *write()* | 29069 | 6341248 | 218.1 |
| all 3 syscalls | 123248 | 13069100 | 106[1] |

*3.2.3 Ring Buffer and Perf Buffer.* The ring buffer and the perf buffer provide the same function for eBPF programs to send data to userspace. As Figure 3-k and Figure 3-l show, for the output operation, the ring buffer achieves better performance. The ring buffer also provides a new reserve+submit method for writing data to it. The eBPF program first performs a reserve operation which returns a pointer to a reserved memory region in the ring buffer, then it can directly write data to the region. Finally, it performs a submit operation to indicate the data is ready to send. The results in Figure 3-k show the reserve+submit operation achieves better performance by reducing memory copying. These results show the ring buffer is a better choice for sending data to userspace compared to the perf buffer.

*3.2.4 Queue and Stack.* As Figure 3-m to Figure 3-p show, the overhead of push operations and pop operations is similar, and the only factor affecting the overhead is the map size. When the cache is hot and the value size is large, we can observe a difference between the overhead of the peek operation and the pop operation. This is because each peek operation accesses the same element, which remains in the CPU cache, resulting in a smaller overhead for copying it.

*3.2.5 Memory Access Overhead of eBPF Map.* Memory access overhead is the main component of eBPF map overhead. It increases with the increase of value size and key size (in the hash map) as well as the savings from the reserve+submit method in the ring buffer, all of which contribute to memory access overhead. However, these overheads become significant only when accessing a large amount of memory, e.g., 1KB and 4KB. In most use cases of eBPF maps, the map's value size (and key size in hash map) is not large, e.g., less than 256 bytes. Therefore, *users do not need to intentionally optimize memory access overhead to improve performance.*

Cache hotness is another important factor affecting the overhead of eBPF map access. As Figure 3 shows, when the value size is less than 256 bytes, the overhead of accessing eBPF maps under cold cache is an order of magnitude higher than when under hot cache. Note that practical BPF programs do not typically run with completely cold/hot caches. Instead, they usually run with a mix of cold/hot cache lines when triggered by workload. We do the benchmarking to study the performance overhead upper/lower bounds with completely cold/hot caches, as shown in Figure 3. To better understand the impact of cache on eBPF performance during workload executions, we conducted another experiment where the eBPF program was triggered by a MySQL workload.

*3.2.6 Benchmark under Synthetic Workload.* To trigger eBPF programs with workload, we started a MySQL service on the benchmark server and implemented a system call tracing application with eBPF. An eBPF program was attached to the entry points of the system calls invoked by the MySQL daemon. Upon syscall invocation, the eBPF program wrote a trace including the syscall ID and the process ID to a ring buffer to send it to userspace. On another server, we ran sysbench as a client to establish a TCP flow to the MySQL service and sent OLTP read SQL queries. When processing these queries, the MySQL daemon invoked the traced system calls and triggered the execution of the attached eBPF program. We selected `pread64()`, `read()`, and `write()` syscalls as tracing targets, which were three most frequently invoked ones by MySQL to perform SQL queries and network I/O.

We conducted four experiments. In the first three experiments, we attached the tracing eBPF program to `pread64()`, `read()`, and `write()` respectively. In the last experiment, we traced all three system calls. Each experiment lasted 5 seconds. As Table 2 shows, the more frequently the eBPF program was executed, i.e., the more times it ran within 5 seconds, the lower the average execution time. This was because the execution frequency of the eBPF program affected the cache hotness when it started to execute. The more frequently the eBPF program is executed, the less other code is executed between two eBPF executions, resulting in fewer flushed eBPF code/data cache lines. We summarize this result as the "*volume discount*" feature of eBPF, which helps eBPF developers to make trade-offs between the number of tracing hooks and overhead in tracing applications.

## 4 CONCLUSION

In this work, we conducted a comprehensive benchmark on eBPF maps to better understand their performance characteristics. Our results indicate that memory footprint and cache hotness are key factors affecting the overhead of eBPF maps among many other factors. We also find that eBPF programs have the volume discount feature. Our benchmark system also supports benchmarking concurrent accesses to eBPF maps, userspace accesses to eBPF maps, and eBPF helper functions. We leave these as the future work.

## ACKNOWLEDGMENTS

C. Liu and L. Wang were supported by the NSFC Project of China under grant 62132009. B. Tak was supported by NRF of Korea grant funded by Korean government (MSIT) (No. NRF-2021R1A5A1021944).

[1]Some might notice this value is lower than the null eBPF program overhead under hot cache in Figure 3. This is because the time in this table is from the kernel's eBPF stats, which only includes the overhead of the eBPF program itself and does not account for the overhead of the trampoline code.

# REFERENCES


[1] Bcc. https://github.com/iovisor/bcc.

[2] bpftrace. https://github.com/bpftrace/bpftrace.

[3] Cilium. https://cilium.io/.

[4] ebpf for windows. https://microsoft.github.io/ebpf-for-windows/.

[5] falco. https://github.com/aquasecurity/tracee.

[6] Katran. https://github.com/facebookincubator/katran.

[7] L4drop: Xdp ddos mitigations. https://blog.cloudflare.com/l4drop-xdp-ebpf-based-ddos-mitigations/.

[8] tracee. https://github.com/aquasecurity/tracee.

[9] ubpf. https://github.com/iovisor/ubpf.

[10] Nadav Amit and Michael Wei. The design and implementation of hyperupcalls. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 97–112, 2018.

[11] Ashish Bijlani and Umakishore Ramachandran. Extension framework for file systems in user space. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 121–134, 2019.

[12] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. hxdp: Efficient software packet processing on fpga nics. *Communications of the ACM*, 65(8):92–100, 2022.

[13] Henri Maxime Demoulin, Isaac Pedisich, Nikos Vasilakis, Vincent Liu, Boon Thau Loo, and Linh Thi Xuan Phan. Detecting asymmetric application-layer {Denial-of-Service} attacks {In-Flight} with {FineLame}. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 693–708, 2019.

[14] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. {BMC}: Accelerating memcached using safe in-kernel caching and pre-stack processing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 487–501, 2021.

[15] Jakub Kicinski and Nicolaas Viljoen. ebpf hardware offload to smartnics: cls bpf and xdp. *Proceedings of netdev*, 1, 2016.

[16] Sebastiano Miano, Xiaoqi Chen, Ran Ben Basat, and Gianni Antichi. Fast in-kernel traffic sketching in ebpf. *ACM SIGCOMM Computer Communication Review*, 53(1):3–13, 2023.

[17] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and KK Ramakrishnan. Spright: extracting the server from serverless computing! high-performance ebpf-based event-driven, shared-memory processing. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 780–794, 2022.

[18] Junxian Shen, Han Zhang, Yang Xiang, Xingang Shi, Xinrui Li, Yunxi Shen, Zijian Zhang, Yongxiang Wu, Xia Yin, Jilong Wang, et al. Network-centric distributed tracing with deepflow: Troubleshooting your microservices in zero code. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 420–437, 2023.

[19] Dave Jing Tian, Grant Hernandez, Joseph I Choi, Vanessa Frost, Peter C Johnson, and Kevin RB Butler. Lbm: A security framework for peripherals within the linux kernel. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 967–984. IEEE, 2019.

[20] Zhe Yang, Youyou Lu, Xiaojian Liao, Youmin Chen, Junru Li, Siyu He, and Jiwu Shu. {$\lambda$-IO}: A unified {IO} stack for computational storage. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 347–362, 2023.

[21] Koen Zandberg, Emmanuel Baccelli, Shenghao Yuan, Frédéric Besson, and Jean-Pierre Talpin. Femto-containers: Lightweight virtualization and fault isolation for small software functions on low-power iot microcontrollers. In *Proceedings of the 23rd ACM/IFIP International Middleware Conference*, pages 161–173, 2022.

[22] Yusheng Zheng, Tong Yu, Yiwei Yang, Yanpeng Hu, Xiaozheng Lai, and Andrew Quinn. bpftime: userspace ebpf runtime for uprobe, syscall and kernel-user interactions, 2023.

[23] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, et al. {XRP}:{In-Kernel} storage functions with {eBPF}. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 375–393, 2022.

[24] Yang Zhou, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu. Electrode: Accelerating distributed protocols with {eBPF}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1391–1407, 2023.