SHORT-PAPER

A Memory Pool Allocator for eBPF Applications

GYUYEONG KIM, Sungshin Women's University, Seoul, South Korea

DONGSU HAN, Korea Advanced Institute of Science and Technology, Daejeon, South Korea

Published: 08 September 2025

SIGCOMM '25: ACM SIGCOMM 2025 Conference
September 8 - 11, 2025
Coimbra, Portugal

Conference Sponsors:
SIGCOMM

https://doi.org/10.1145/3748355.3748370
ISBN: 9798400720840

A Memory Pool Allocator for eBPF Applications

Gyuyeong Kim
Sungshin Women's University
Seoul, Republic of Korea
gykim@sungshin.ac.kr

Dongsu Han
KAIST
Daejeon, Republic of Korea
dhan.ee@kaist.ac.kr

ABSTRACT

eBPF enables high-performance kernel-level execution by eliminating networking stack traversal and context switching. Despite the advantages, eBPF applications face strict memory management constraints due to the eBPF verifier requirements that mandate static memory allocation. This limitation imposes a fundamental tradeoff between application performance and memory efficiency, ultimately restricting the potential of eBPF. We present Kerby, a dynamic memory pool allocator for eBPF that enables eBPF applications to dynamically manage pre-allocated memory by representing variable-length data as collections of fixed-size blocks. This allows applications to increase the amount of kernel-resident data while minimizing internal fragmentation. Our preliminary evaluation with key-value store implementations demonstrates that Kerby achieves significant improvements in both memory utilization and throughput.

CCS CONCEPTS

• **Software and its engineering** → **Memory management**; • **Networks** → **In-network processing**;

KEYWORDS

eBPF, memory pooling, operating systems

ACM Reference Format:
Gyuyeong Kim and Dongsu Han. 2025. A Memory Pool Allocator for eBPF Applications. In *3rd Workshop on eBPF and Kernel Extensions (eBPF '25), September 8–11, 2025, Coimbra, Portugal.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3748355.3748370

1 PROBLEM AND MOTIVATION

Extended Berkeley Packet Filter (eBPF) [1] has emerged as a powerful framework for extending OS kernel functionality without modifications [6], enabling high-performance applications through kernel-level execution. By offloading application logic to the kernel, eBPF applications can bypass networking stack traversal and eliminate user-kernel context switching. This approach has demonstrated significant benefits across various domains [13], including key-value stores [9, 15, 17], lock and log managers [17], consensus protocols [16], and sketch [11].

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

ACM ISBN 979-8-4007-2084-0/25/09.
https://doi.org/10.1145/3748355.3748370

Despite its potential, the strict constraints of the eBPF verifier [14] limit the efficiency of eBPF applications. In particular, eBPF applications are constrained to use static memory allocation because the eBPF program must be verified statically before execution. This limitation creates a fundamental trade-off between performance and memory efficiency, especially for applications handling variable-length data.

This trade-off is clearly illustrated in existing in-kernel key-value stores [9, 17], which are restricted to fixed-size items because BPF map entry slots must have predetermined sizes. System designers face an unappealing choice: either allocate large slots to maximize in-kernel data processing but risk substantial internal fragmentation with small items, or use smaller slots to reduce fragmentation but force most data into user space, sacrificing performance. This dilemma leads us to ask: *how can we achieve high performance and memory efficiency simultaneously for eBPF applications?*

2 KERBY APPROACH

Our goal is to make eBPF applications achieve high performance and memory efficiency at the same time. Existing OS memory allocators like the buddy allocator or the slab allocator cannot be applied since the number of entries in BPF maps cannot be rebalanced. To achieve the goal, we propose Kerby, a memory pool allocator for eBPF. The idea is to pool memory space for eBPF applications and manage it dynamically. Kerby divides the pre-allocated memory pool into multiple fixed-size blocks, and these blocks are dynamically combined to represent variable-length data. For example, a 1024-B object uses eight 128-B blocks.

The memory pool allocation of Kerby improves memory efficiency for kernel-resident data, enabling eBPF applications to process more requests at higher performance. By limiting internal fragmentation to only the last allocated block per object, it minimizes memory waste. It also maintains mainline kernel compatibility by operating within eBPF verifier constraints. For latency-sensitive applications, Kerby offers an ideal solution that combines the flexibility of dynamic memory allocation with low operational overhead.

Challenges. A key challenge is to track the list of free blocks, especially reclaimed ones, as these blocks should be reused. Strawman solutions include having a queue or a bitmap-based list for reclaimed blocks. However, these solutions complicate making write operations atomic. In many applications, we need to lock a data entry to handle concurrent requests to the same entry. Unfortunately, the eBPF verifier does not allow BPF programs to call functions while holding a lock, though we need to call BPF helper functions to update the free block list during a write operation.

Design. To address this, we adopt a counter-intuitive approach: we do not maintain a free block list at all. Instead, we use each block index only once by ensuring the block index is monotonically

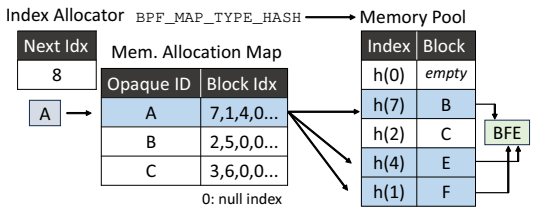

Figure 1: Example of read operations in Kerby.

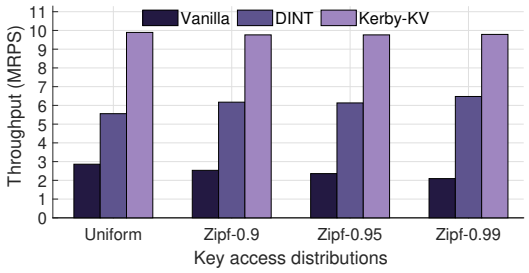

Figure 2: Throughput with different skewness.

increasing. The block index is 64 bits wide, providing a large upper bound of $2^{64} - 1$. The memory pool computes a hash of the block index to access the block entries, keeping the effective index within the memory pool bounds. Kerby consists of three core components as follows.

1) Memory allocation map translates each opaque ID to its corresponding block indices in the pool. The opaque ID represents an object identifier, such as a key, object sequence number, or item index. The map can be a BPF hash map (`BPF_MAP_TYPE_HASH`), a BPF array map (`BPF_MAP_TYPE_ARRAY`), and per-CPU maps (e.g., `BPF_MAP_TYPE_PERCPU_ARRAY`), depending on the required ID size of the target applications. For example, the key-value store should use a BPF hash map since item keys are typically larger than 4 bytes, which is the maximum identifier size supported by a BPF array map. We can configure the number of indices and the block size depending on the applications. **2) Memory pool** is a pre-allocated memory divided into fixed-size blocks. We use a BPF hash map because it has an internal collision resolution mechanism, and its hash index ensures each block index is mapped to a single block only. **3) Index allocator** is a BPF array map with a single entry that represents a monotonically-increasing block index. We use the `__atomic_fetch_add` instruction to issue new block indices atomically, eliminating concurrency issues.

Kerby supports concurrent (de)allocation of variable-size objects in the following way. When allocating a new object, Kerby allocates the desired number of blocks from the memory pool using indices retrieved from the index allocator. Kerby stores the ordered list of indices in the memory allocation map for the opaque ID, which is the data handle for the application. When an object is deallocated, Kerby first removes the object entry from the allocation map. After that, Kerby clears each block used by the object in the memory pool, using eBPF delete. Kerby also supports object resizing by only (de)allocating blocks in the memory pool. Figure 1 shows an example of how the data `BFE` is reconstructed. 0 is the null index.

Kerby-KV. To demonstrate the applicability of Kerby, we implement Kerby-KV, an in-kernel key-value store built on top of Kerby. The opaque ID in the allocation map is the item key, and the blocks in the memory pool are a part of the item value. Kerby-KV stores metadata like the value length and a spinlock for each key in the allocation map. Kerby-KV supports keys up to 128 bytes. The allocation map contains up to eight indices to 128-B blocks in the memory pool. Note that the supported item size is configurable.

Preliminary results. To demonstrate the efficiency of Kerby, we compare Kerby-KV with the vanilla Linux and DINT [17], which relies on static memory allocation. The value slot size is set to 128 bytes to ensure 100% memory efficiency for DINT. Experiments are performed with 8 clients and 1 server with Linux kernel 6.8.0 under various key access distributions. We measure read throughput using a mix of 75% 128-B and 25% 1024-B values. All keys are 8 bytes to match the supported key size by DINT. We modified DINT to allow it to handle oversized items in user-space.

Figure 2 shows the results. We can see that Kerby-KV outperforms Vanilla and DINT significantly. This is because DINT handles oversized 1024-B items in the user space after experiencing burdensome kernel stack overheads, whereas Kerby-KV handles both 128-B and 1024-B items fully in the kernel with dynamic memory pool allocation, achieving high performance. The reason why the performance of DINT increases as the skewness grows is that 128-B items are accessed more.

3 DISCUSSION AND RELATED WORK

Usecases. Kerby can benefit applications beyond key-value stores, especially those handling variable-length data. For example, log managers also benefit from Kerby by supporting variable-length logs, unlike existing eBPF implementations restricted to fixed-sized entries [17]. This flexibility proves valuable in distributed transactions where log entry sizes vary with complexity.

Alternative approaches. There are a few recent efforts for dynamic memory allocation in eBPF, but they provide only limited support. The `bpf_obj_new` [4] enables the creation of objects at run time but only for fixed-size objects defined at compile-time, making it hard to handle variable-length data. BPF Arena [2, 3] allows memory sharing between BPF programs and userspace applications, with memory allocated on demand. However, it currently cannot allocate/deallocate memory at the XDP hook [7, 10] since it requires a sleepable context [5] while the XDP runs in an atomic (i.e., non-sleepable) context. It may cause internal fragmentation as it allocates memory in pages (i.e., 4 KB). It also supports memory only up to 4 GB. KFlex [8] is a kernel extension framework that includes extension heaps for dynamic allocation, but its memory feature is hard to integrate with the mainline kernel due to different safety check principles [12].

ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their insightful comments and constructive feedback. This research was sponsored by the National Research Foundation of Korea (NRF) grants funded by the Ministry of Science and ICT (No. RS-2025-00522990 and RS-2024-00340099). Gyuyeong Kim is the corresponding author.

REFERENCES

[1] 2024. eBPF. https://ebpf.io/. (2024).

[2] 2024. A proposal for shared memory in BPF programs. https://lwn.net/Articles/961941/. (2024).

[3] 2025. A look at what's possible with BPF arenas. https://lwn.net/SubscriberLink/1019885/7aebede8a200949b/. (2025).

[4] 2025. SCX eBPF macro bpf_obj_new. https://docs.ebpf.io/ebpf-library/scx/bpf_obj_new/. (2025).

[5] 2028. Atomic context and kernel API design. https://lwn.net/Articles/274695/. (2028).

[6] Theophilus A. Benson, Prashanth Kannan, Prankur Gupta, Balasubramanian Madhavan, Kumar Saurabh Arora, Jie Meng, Martin Lau, Abhishek Dhamija, Rajiv Krishnamurthy, Srikanth Sundaresan, Neil Spring, and Ying Zhang. 2024. NetEdit: An Orchestration Platform for eBPF Network Functions at Scale. In *Proc. of ACM SIGCOMM*. Association for Computing Machinery, New York, NY, USA, 721–734. https://doi.org/10.1145/3651890.3672227

[7] Zhongjie Chen, Qingkai Meng, ChonLam Lao, Yifan Liu, Fengyuan Ren, Minlan Yu, and Yang Zhou. 2025. eTran: Extensible Kernel Transport with eBPF. In *Proc. of USENIX NSDI*. USENIX Association.

[8] Kumar Kartikeya Dwivedi, Rishabh Iyer, and Sanidhya Kashyap. 2024. Fast, Flexible, and Practical Kernel Extensions. In *Proc. of ACM SOSP*. Association for Computing Machinery, New York, NY, USA, 249–264. https://doi.org/10.1145/3694715.3695950

[9] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. 2021. BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing. In *Proc. of USENIX NSDI*. USENIX Association, 487–501.

[10] 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 *Proc. of ACM CoNEXT*. Association for Computing Machinery, New York, NY, USA, 54–66. https://doi.org/10.1145/3281411.3281443

[11] Sebastiano Miano, Xiaoqi Chen, Ran Ben Basat, and Gianni Antichi. 2023. Fast In-kernel Traffic Sketching in eBPF. *SIGCOMM Comput. Commun. Rev.* 53, 1 (April 2023), 3–13.

[12] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. 1996. Dealing with disaster: surviving misbehaved kernel extensions. In *Proc. of USENIX OSDI*. Association for Computing Machinery, New York, NY, USA, 213–227. https://doi.org/10.1145/238721.238779

[13] Farbod Shahinfar, Sebastiano Miano, Giuseppe Siracusano, Roberto Bifulco, Aurojit Panda, and Gianni Antichi. 2023. Automatic Kernel Offload Using BPF. In *Proc. of ACM HotOS*. Association for Computing Machinery, New York, NY, USA, 143–149.

[14] Hao Sun and Zhendong Su. 2024. Validating the eBPF Verifier via State Embedding. In *Proc. of USENIX OSDI*. USENIX Association, Santa Clara, CA, 615–628. https://www.usenix.org/conference/osdi24/presentation/sun-hao

[15] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. 2022. XRP: In-Kernel Storage Functions with eBPF. In *Proc. of USENIX OSDI*. USENIX Association, Carlsbad, CA, 375–393.

[16] Yang Zhou, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu. 2023. Electrode: Accelerating Distributed Protocols with eBPF. In *Proc. of USENIX NSDI*. USENIX Association, Boston, MA, 1391–1407. https://www.usenix.org/conference/nsdi23/presentation/zhou

[17] Yang Zhou, Xingyu Xiang, Matthew Kiley, Sowmya Dharanipragada, and Minlan Yu. 2024. DINT: Fast In-Kernel Distributed Transactions with eBPF. In *Proc. of USENIX NSDI*. USENIX Association, Santa Clara, CA, 401–417.