Pairwise BPF Programs Should Be Optimized Together

Milo Craun
Virginia Tech
Blacksburg, VA, USA
miloc@vt.edu

Dan Williams
Virginia Tech
Blacksburg, VA, USA
djwillia@vt.edu

ABSTRACT

BPF programs are extensively used for tracing and observability in production systems where performance overheads matter. Many individual BPF programs do not incur serious performance degrading overhead on their own, but increasingly more than a single BPF program is used to understand production system performance. BPF deployments have begun to look more like distributed applications; however, this is a mismatch with the underlying Linux kernel, potentially leading to high overhead cost. In particular, we identify that many BPF programs follow a pattern based on *pairwise program deployment* where entry and exit probes will be attached to measure a single quantity. We find that the pairwise BPF program pattern results in unnecessary overheads. We identify three optimizations—*BPF program inlining*, *context aware optimization*, and *intermediate state internalization*—that apply to pairwise BPF programs. We show that applying these optimizations to an example pairwise BPF program can reduce overhead on random read throughput from 28.13% to 8.98% and on random write throughput from 26.97% to 8.60%. We then examine some key design questions that arise when seeking to integrate optimizations with the existing BPF system.

CCS CONCEPTS

• **Software and its engineering** → **Operating systems**; **Software performance**; **Software testing and debugging**;

KEYWORDS

eBPF, pairwise BPF, dynamic tracing, observability

ACM Reference Format:
Milo Craun and Dan Williams. 2025. Pairwise BPF Programs Should Be Optimized Together. In *3rd Workshop on eBPF and Kernel Extensions (eBPF '25), September 8–11, 2025, Coimbra, Portugal.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3748355.3748362

1 INTRODUCTION

BPF is a Linux kernel subsystem that allows verified safe kernel extensions to be deployed. Since its introduction, BPF has seen increasing usage in a variety of domains from high performance networking [12, 17] to security [4] to observability [1–3, 9].

BPF program cost is important when deploying programs on production systems because it can directly impact quality of service. At the same time, deploying BPF tracing programs on production systems is valuable for performance engineers and system operators, leading to a trade-off between maintaining production performance and operator power.

Increasingly, BPF programs are part of a distributed deployment, where a single application consists of multiple BPF programs and user-space components that all communicate together. In this work we identify a common pattern of BPF tracing program: *pairwise BPF programs*. Pairwise BPF programs consist of an entry and an exit probe that work together to perform a tracing operation. Pairwise programs thus provide a clear and compelling example of BPF programs as distributed applications.

Unfortunately, we observe that the structure of pairwise BPF programs leads to unnecessary overhead cost. The entry and exit programs are attached to locations in the kernel that are executed closely together, but they must still pay the full cost of hooking twice, one for each. Both probes may also contain duplicate context checking code, which is needed by the BPF verifier but is redundant in scope between both probes. Finally, they must explicitly export and import intermediate state through the use of BPF maps, which can be an expensive operation.

To address these overheads, we describe three performance optimizations for pairwise BPF programs, each targeting one of the above identified sources of overhead. By manually implementing all optimizations we show a reduction of overhead on random read throughput from 28.13% to 8.98% and on random write throughput from 26.97% to 8.60%. We then describe a series of key design questions for how to create a system that safely and automatically applies our optimizations to real-world pairwise BPF programs in production.

2 COST OF DISTRIBUTED BPF PROGRAMS

BPF tracing and observability programs follow a distributed model where they are broken into components that must explicitly share state and communicate over channels. However, the Linux kernel is not a distributed system, meaning there is a mismatch between BPF program deployment and the underlying system. In this section we show that the mismatch causes additional overhead for BPF programs in tracing use cases, by identifying the *pairwise BPF program* pattern and analyzing the performance of an example pairwise program.

2.1 BPF Tracing Programs Are Distributed

Typically, a single BPF tool consists of one or more BPF programs, along with one or more user-space components. BPF programs mainly communicate with user-space by explicitly using BPF maps to share data. BPF maps are also used to communicate between different BPF programs. The combination of components make up

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

ACM ISBN 979-8-4007-2084-0/25/09.
https://doi.org/10.1145/3748355.3748362

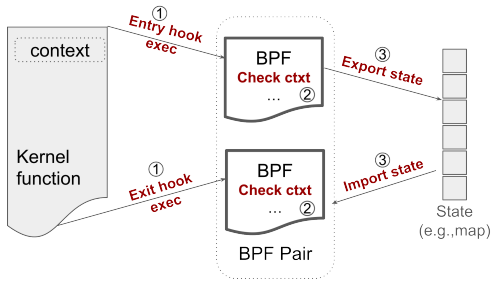

Figure 1: Pairwise BPF execution model

a single distributed application, but all are treated as separate BPF objects.

Through a study of libbpf-tools [8], we identified a pervasive pattern of distributed BPF tracing programs called *pairwise BPF programs*, or those with a related entry and exit component. We found that 22 out of 57 tools contained pairwise BPF programs, indicating that this is a common pattern for BPF tracing programs. Studying pairwise BPF programs provides insight into the general performance consequences of treating BPF programs as a distributed application. The rest of this section describes pairwise BPF and presents a study of the overheads associated with pairwise BPF programs.

2.2 Pairwise BPF Programs

In this subsection we describe the pairwise BPF program pattern by connecting a general model of pairwise BPF program execution, shown in Figure 1, and an example BPF program from libbpf-tools: `fsdist.bpf.c` [6] shown in Figure 2. We now walk through each circled label in Figure 1.

(1) Pairwise BPF programs consist of two separate, but semantically linked BPF programs. The first BPF program is attached to an *entry hook* and is executed before a kernel function is run or an event is completed, while the second BPF program is attached to an *exit hook* and is executed after the kernel function returns or the event is handled. In the `fsdist` BPF program, Figure 2, the entry program is called `probe_entry` and the exit program is called `probe_return`. The loader for this program takes a filesystem as an argument and attaches the programs to up to five corresponding kernel functions. The entry and exit programs are attached using *fentry* and *fexit* hookpoints if supported, and the slower *kprobe* and *kretprobe* hookpoints otherwise. Different hookpoints have different costs (which we will show in Figure 3). For the rest of the example, we will assume that the program is used to trace an `ext4` filesystem, and that it is attached to *fentry/fexit*.

(2) Each of the entry and exit BPF programs must perform context checks at the beginning of their execution. The main purpose of the context check is to collect any needed information for the execution of the rest of the program, as well as resolve any dynamic configuration information. In the `fsdist` BPF program, the context checks for the entry program involve reading the current `pid` and comparing it against a `target_pid` (lines 3-4,7-8). The exit program ensures that the file system `op` value is within bounds and

```
1  static int probe_entry()
2  {
3    __u64 pid_tgid = bpf_get_current_pid_tgid();
4    __u32 pid = pid_tgid >> 32;
5    __u32 tid = (__u32)pid_tgid;
6    __u64 ts;
7    if (target_pid && target_pid != pid)
8      return 0;
9    ts = bpf_ktime_get_ns();
10   bpf_map_update_elem(&starts, &tid, &ts, BPF_ANY);
11   return 0;
12 }
```

```
1  static int probe_return(enum fs_file_op op)
2  {
3    __u32 tid = (__u32)bpf_get_current_pid_tgid();
4    __u64 ts = bpf_ktime_get_ns();
5    __u64 *tsp, slot;
6    __s64 delta;
7    tsp = bpf_map_lookup_elem(&starts, &tid);
8    if (!tsp)
9      return 0;
10   if (op >= F_MAX_OP)
11     goto cleanup;
12   delta = (__s64)(ts - *tsp);
13   if (delta < 0)
14     goto cleanup;
15   if (in_ms)
16     delta /= 1000000;
17   else
18     delta /= 1000;
19   slot = log2l(delta);
20   if (slot >= MAX_SLOTS)
21     slot = MAX_SLOTS - 1;
22   __sync_fetch_and_add(&hists[op].slots[slot], 1);
23 cleanup:
24   bpf_map_delete_elem(&starts, &tid);
25   return 0;
26 }
```

Figure 2: `fsdist.bpf.c` code snippet, with entry and return probes. Highlights indicate context checks and externalized state

additionally reads an option to see if the resulting data should be presented in milliseconds or not (lines 10-11, 15-18).

(3) A key responsibility of the entry probe is to export state for the exit probe to use later. For BPF programs this will typically involve using a BPF map to store some information. The `fsdist` entry program records a timestamp, and then stores it into a BPF hash map, using the `tid` as a key (lines 5, 9-10). The exit probe will then import external state for use in the program, involving a map lookup and required safety checks. The `fsdist` exit program reads the timestamp by looking up the `tid` in a BPF map. The value is then ensured to be non-null so the program can safely access the data. The program then uses the original timestamp to calculate how long the filesystem operation took. At the end of the exit program the timestamp map entry for the `tid` is deleted to allow for measurement of future filesystem operations from the same thread (lines 3, 7-9, 24).

Configuration	Random Read (MB/sec)	Random Write (MB/sec)
Baseline	2251.95 ± 27.53	1718.21 ± 48.35
`fsdist`	1618.46 ± 16.96 (28.13%)	1254.69 ± 47.47 (26.97%)

Table 1: Random read and write performance with `fsdist` BPF program. Standard deviation is shown as error, and percentage overhead is in parentheses.

2.3 Cost of Pairwise BPF Programs

All BPF programs come with overhead costs, but pairwise programs potentially come with more costs, as they involve paying hooking costs twice, involve extra context checks, and involve more map accesses for exporting and importing state. In this subsection we present the three main sources of pairwise program overhead. We performed two experiments to quantify and understand the costs of pairwise BPF programs. The first experiment evaluates the overhead of `fsdist` in a real world test, while the second experiment seeks to understand the cost of hookpoints. The result of our experiments show that pairwise BPF programs can cause overhead of up to 28.14%. We provide qualitative arguments for other sources of overhead we did not measure.

Overall Cost: To measure the performance overhead of `fsdist` we performed an experiment comparing IOzone [7] random read and random write performance with and without `fsdist` deployed. Our system is a Linux kernel 6.13 virtual machine running IOzone with one process on an `ext4` filesystem. We repeat each trial 10 times and present the mean in Table 1. We find that using `fsdist` incurs a 28.13% overhead on random read performance and a 26.97% overhead on random write performance. The key takeaway is that even simple pairwise BPF programs can generate high overheads.

Hooking Cost: Pairwise BPF programs involve twice as many hooks as a single BPF program, meaning the impact of hookpoint cost is doubled. To quantify the cost of hooking we performed an experiment where we measure the runtime of an `openat` system call while empty BPF programs are attached to different hookpoints. `Fentry`, optimized kprobe, and raw tracepoint hookpoints are specially designed for maximum performance. Tracepoints present a more stable interface for tracing, but are slower. Kprobe is a less optimized hookpoint, but it can attach to anywhere in the kernel, making it flexible.

As the BPF programs immediately return, all overhead is only from the cost of hooking and returning from the BPF program. We present our results in Figure 3. We find that the overhead on the `openat` system call is between ~19%, for `fentry` and optimized kprobes, and ~119% for kprobes. For pairwise BPF programs the hooking cost will be paid twice. From this experiment we conclude that the cost of hooking can be a significant part of the overall performance overhead of pairwise BPF programs.

Context Checks: BPF programs require checks on variables to ensure that they can pass the BPF verifier and are safe. In addition to checks, there may be additional code that is used to set up the BPF program. As the entry and exit probes are separate, they must pass the verifier independently. Doing so may result in redundant checks and code which can cause additional overheads.

Externalized State: Pairwise BPF programs require sharing state between entry and exit probes. The primary mechanism is to use a BPF map to provide communication between the two programs. Depending on the map type and parameters, updating and accessing maps can cause significant overhead [15]. The pair of programs do both, and the use of BPF maps for sharing state can cause overhead.

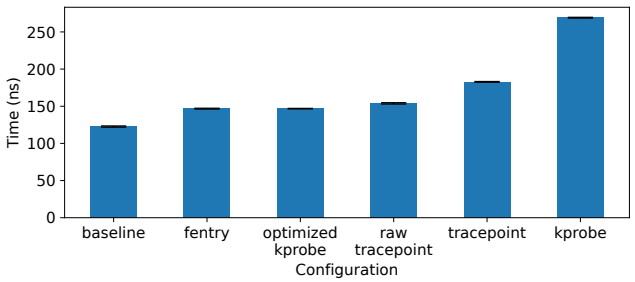

Figure 3: Hookpoint cost for different hookpoints.

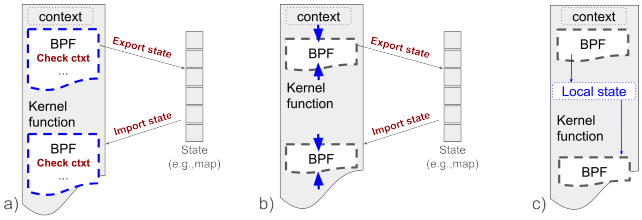

Figure 4: The three optimizations we present: BPF program inlining, context aware optimization (CAO), and intermediate state internalization.

3 OPTIMIZING PAIRWISE BPF PROGRAMS

From our performance study, we found that pairwise BPF programs can impose high overhead costs. Next, we analyze the features of pairwise BPF programs and present three optimizations, BPF program inlining, context aware optimization (CAO), and intermediate state internalization (ISI), shown in Figure 4, that can potentially improve the performance of pairwise BPF programs. After describing each optimization, we walkthrough the process of manually applying all optimizations to the `fsdist` BPF program. Finally, we quantify by how much the use of our optimizations reduces overhead.

3.1 BPF Program Inlining

We present BPF program inlining as a method to reduce the hooking cost, and to enable our other optimizations. As shown in Figure 4(a), performing inlining involves directly writing BPF program text into the kernel text where it is attached. Doing so removes the hooking cost, and also places entry and exit programs into the same context.

3.2 Context Aware Optimization

As shown in Figure 4(b), context aware optimization (CAO) involves removing unnecessary context checks from pairwise BPF programs, and linking together kernel code with BPF program code. One example is if kernel code fetches a large data structure which is

then used by the BPF program. Without CAO, the BPF program would have to re-fetch the data structure, but by applying CAO, the BPF program can use the already available data.

Not all BPF programs will benefit from CAO. For example, a BPF program that does not make use of any hookpoint specific information, such as specific kernel data structures, likely will not benefit from CAO, as it is hookpoint context independent by nature.

3.3 Intermediate State Internalization

A large opportunity for optimization in pairwise BPF programs takes the form of internalizing intermediate state. As described in §2, pairwise BPF programs must explicitly externalize state through BPF maps. As shown in Figure 4 (c), intermediate state internalization (ISI) improves performance by avoiding the usage of BPF maps, and by removing code that supports externalization of state.

In general, BPF map performance can be an expensive part of BPF programs [15]. By treating the pair of BPF programs as a single semantic unit and putting them in the same context we can replace the usage of globally shared state (i.e. BPF maps) with much faster, but local and private state (i.e. a stack variable or thread local storage).

The two BPF programs may also include additional supporting code needed for importing and exporting intermediate state. For example, both the entry and exit may need to fetch common data, such as a `pid`, `tid`, or a common value from a map. With ISI, we can eliminate the need for fetching this data if it is only used to support exporting and importing state. If the data has other uses, we can potentially eliminate a re-fetch of the data, if we are sure that the data will not change across the kernel function execution.

3.4 Optimization Example

In this section we walk through the process of hand optimizing the BPF program in Figure 2. The program originally attached to five different filesystem functions. We manually optimized all five programs independently. We only describe the process specifically for the `ext4_file_read_iter` kernel function, as it is representative of all five programs. We present the final optimized code in Figure 5.

Inlining: The first step is to inline the BPF program code at the beginning and end of the traced function, shown in Figure 4 (a). We do this by directly inserting the code at the entry and exit of the traced kernel function. Then we rewrite the control flow of the kernel function in order to ensure that the BPF program is called. We had to rewrite all early return paths to direct control flow to the return probe by storing return values, and performing unconditional jumps to the return probe code.

CAO: Next we performed CAO, shown in Figure 4 (b). In the `probe_return` on line 10 and 22, there is a reference to an `op` variable. At this point we know what the value of `op` is because it is known based on the attachment location. In particular we know that `op` is `F_READ`. Related to this, we also resolve dynamic configuration checks. In the original program on lines 7 of the `probe_entry`, and lines 10 and 15 of the `probe_return` there is code that consists of checks against configuration values and a check for the `op` field. We know the values of each of these ahead of time, and we remove checks, as well as ensure that the value of `op` is in range.

```
1  static ssize_t ext4_file_read_iter(struct kiocb *iocb, struct
       iov_iter *to)
2  {
3    ssize_t ret;
4    u64 ts = ktime_get_ns();
5    struct inode *inode = file_inode(iocb->ki_filp);
6    if (unlikely(ext4_forced_shutdown(inode->i_sb))) {
7      ret = -EIO;
8      goto file_read_iter_exit;
9    }
10   if (!iov_iter_count(to)) {
11     ret = 0;
12     goto file_read_iter_exit;
13   }
14   #ifdef CONFIG_FS_DAX
15   if (IS_DAX(inode)) {
16     ret = ext4_dax_read_iter(iocb, to);
17     goto file_read_iter_exit;
18   }
19   #endif
20   if (iocb->ki_flags & IOCB_DIRECT) {
21     ret = ext4_dio_read_iter(iocb, to);
22     goto file_read_iter_exit;
23   }
24   ret = generic_file_read_iter(iocb, to);
25 file_read_iter_exit:
26   u64 delta = ts - ktime_get_ns();
27   if (delta < 0)
28     return ret;
29   delta /= 1000;
30   u64 slot = fsdist_log2l(delta);
31   if (slot >= FSDIST_MAX_SLOTS)
32     slot = FSDIST_MAX_SLOTS - 1;
33   __sync_fetch_and_add(&hists[F_READ].slots[slot], 1);
34   return ret;
35 }
```

Figure 5: Model optimized version of `fsdist.bpf.c` for the `ext4_file_read_iter` function. Highlights indicate which optimization: inlining, CAO, and ISI

ISI: Now, we are able to internalize state, shown in Figure 4 (c). The `starts` BPF hash map is no longer needed to pass the timestamp between the `probe_entry` and `probe_return`, as they are now inside the same function scope. We instead store an initial time stamp on the stack, and reference that when we compute the overall time delta. Doing so allows us to remove all BPF map access helpers related to the starts map, as it is no longer used. After this we remove any supporting code and state needed to support the usage of the starts map. In this case we remove all code associated with collecting a `tid` to index the starts map. Depending on whether or not the `target_pid` configuration was set we can also remove accessors to `pid`.

3.5 Optimization Performance Analysis

In this subsection we analyze the performance of the hand-optimized `fsdist` pairwise BPF program. As before, we use IOzone to benchmark random read and write throughput. We use Linux kernel 6.13 with different sets of optimizations applied. We include a baseline with an unmodified kernel and a test with the regular BPF program deployed. We then perform inlining followed

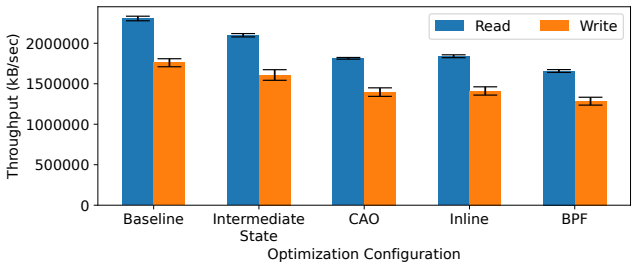

Figure 6: IOzone read/write throughput.

by CAO followed by ISI, as described above for all five attachment points. Each optimization is applied after the previous had also been applied. We present the results in Figure 6.

Overall we find that applying all three optimizations reduced overhead on random read throughput from 28.13% to 8.98% and on random write throughput from 26.97% to 8.60%. There are three features of note in the data. First, we see that there is a jump from the BPF performance to the strictly inlined version, indicating that the hooking overhead does introduce overhead outside of micro benchmarks. Second, we see that the performance between the CAO and inlining vs inlining alone is not significant. This is expected because the example program does not contain many opportunities for further CAO, so any performance difference is likely obscured due to experimental noise. We expect that there are other less context independent programs which will receive a greater benefit from CAO. Finally, we see that there is another step up in performance when we apply ISI. From the performance experiment we conclude that there is performance benefits from the optimizations we describe.

4 APPLYING OPTIMIZATIONS

In this section we raise two key design questions and explore them. We first ask: *How much optimization can be automatic?* Automatic optimization refers to how much pairwise BPF programs can be optimized without any modification to the programs. We then ask: *What are the mechanisms for applying optimizations?*

4.1 Automatic Optimization

A key challenge to automatically applying optimizations is to be able to programmatically identify pairwise BPF programs, and their features. The kernel could use metadata about programs such as where they will attach and the order of BPF program loading, to help reason about when two programs are pairwise. The kernel could also look for access patterns to maps in order to identify when a BPF map is being used for externalizing state. For example, if after identifying a pairwise BPF program, one map is written to only by the entry probe, and is read only by the exit probe, then that map is likely used for externalizing state. Additionally, the kernel must be able to identify what code is only used to support state externalization.

However, the above heuristics may be inaccurate and may not be able to identify all opportunities for optimization. The key takeaway is that fully automating all optimization is hard, and likely to result in missing optimization opportunities.

The question then becomes how much does the BPF program developer have to do to enable optimizations. One approach could be to provide a framework for the developer to write programs in such a way as to avoid ever externalizing state or putting in redundant code. In this scenario, no code would need to be removed from the program, and the only automatic optimization would be inlining the program code in the kernel. Another approach could be to simply include what maps are shared between entry and exit probes as metadata, and then rewrite the program automatically.

4.2 Implementing Optimizations

Another design question is the mechanisms to use for implementing optimizations. A natural approach to supporting all optimizations is to use dynamic function rewriting to essentially automate the optimization process described in §3.4. This would allow for all optimizations to be performed, and would provide flexibility for additional optimizations. However, rewriting kernel functions dynamically is invasive, and carries challenges such as ensuring that kernel function safety is not compromised and overcoming general concerns about kernel text modification. Additionally, hookpoints and BPF subsystem code ensure an initial state before BPF program execution, such as disabling migration or preemption. As part of inlining, the additional setup must be addressed by either ensuring that it is safe to remove, or by implementing it inline with the BPF code.

Another approach is to make pairwise BPF programs first class objects in the BPF subsystem, by introducing a special type of local BPF map, new hookpoint, and verifier modifications. Pairwise programs could then be written, or automatically converted as discussed above, to use the new BPF mechanisms to be more efficient. The full BPF inlining and CAO benefits may not be present with this approach, but a combination approach could be taken to minimize kernel text modification, while maximizing optimization opportunity.

5 RELATED WORKS

In this section we describe other work that seeks to optimize BPF program performance or otherwise reduce the impact of BPF programs on running systems.

Optimize BPF Program Cost: Several pieces of work seek to optimize BPF program code itself. K2 [18] is an optimizing compiler that works to synthesize that is more performant as well as being amenable to the verifier constraints on BPF programs. Another optimizing compiler for BPF is Merlin [16]. Merlin works to optimize BPF programs through special instruction merging and strength reduction techniques. Both of these projects try to optimize BPF bytecode before verification through the use of traditional compiler techniques. They do not include optimizations for pairs of BPF programs, instead focusing on individual programs only.

Optimizing Hookpoints: Much work has been done in the Linux kernel to reduce the costs of hookpoints. Raw tracepoints [5] and fentry/fexit [10] hooks are more efficient than traditional tracepoints or optimized kprobes. Both optimized hookpoints lose out on some generality in order to improve performance. Raw tracepoints do not save arguments in the context for BPF programs,

while fentry/fexit can only be used on function entry and exit. Another work places nop instructions into the kernel to allow more kprobes to be optimized to avoid the cost of a double trap [13]. Unlike optimizing hookpoints, we use semantic information about BPF programs to perform optimizations that are useful for our specific case. One interesting line of research would be to see if a new hookpoint mechanism can be developed specifically for supporting pairwise BPF programs.

Dynamically Optimize: Some work seeks to improve the performance of BPF dynamically. KFuse [14] aims to optimize chains of BPF programs by merging programs together. Doing so allows for BPF programs installed by different users to be combined together. Craun et. al propose the implementation of per-process BPF programs in order to reduce overhead for processes that do not need to be traced [11]. Both projects use dynamic mechanisms to try to reduce the overall overhead of BPF programs on the system. The usage of per-process kernel views in [11] represents a use of kernel binary modification, which is related to our idea of dynamic kernel function rewriting.

6 SUMMARY

In conclusion, pairwise BPF programs are a common pattern found in dynamic tracing BPF use cases and are indicative of the kinds of additional overheads that can come from programming BPF programs to be distributed, while running them in a centralized way. Despite being common, a lack of support leads to high performance overheads from deploying pairwise BPF programs. We identified three optimizations and show that the application of them can significantly reduce the overall overhead of pairwise BPF programs. Finally, we presented some preliminary design work indicating potential ways to apply the optimizations in a real-world system.

7 ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments. This work was funded in part by NSF grant CNS-2236966.

8 ETHICAL CONCERNS

This work raises no ethical concerns.

REFERENCES

[1] 2024. bcc. https://github.com/iovisor/bcc. (May 2024).
[2] 2024. bpftrace. https://github.com/bpftrace/bpftrace. (May 2024).
[3] 2024. Cilium. https://cilium.io. (May 2024).
[4] 2024. Falco. https://falco.org/. (December 2024).
[5] 2024. Program type BPF_PROG_TYPE_RAW_TRACEPOINT. https://docs.ebpf.io/linux/program-type/BPF_PROG_TYPE_RAW_TRACEPOINT/. (November 2024).
[6] 2025. fsdist.bpf.c. https://github.com/iovisor/bcc/blob/master/libbpf-tools/fsdist.bpf.c. (May 2025).
[7] 2025. IOzone Filesystem Benchmark. https://www.iozone.org/. (May 2025).
[8] 2025. libbpf-tools. https://github.com/iovisor/bcc/blob/master/libbpf-tools/. (May 2025).
[9] 2025. pixie. https://px.dev/. (May 2025).
[10] 2025. Program type BPF_PROG_TYPE_TRACING. https://docs.ebpf.io/linux/program-type/BPF_PROG_TYPE_TRACING/. (March 2025).
[11] Milo Craun, Khizar Hussain, Uddhav Gautam, Zhengjie Ji, Tanuj Rao, and Dan Williams. 2024. Eliminating eBPF Tracing Overhead on Untraced Processes. In *Proceedings of the ACM SIGCOMM 2024 Workshop on EBPF and Kernel Extensions (eBPF '24)*. Association for Computing Machinery, New York, NY, USA, 16–22. https://doi.org/10.1145/3672197.3673431
[12] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. 2018. The eXpress data path: fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '18)*. Association for Computing Machinery, New York, NY, USA, 54–66. https://doi.org/10.1145/3281411.3281443
[13] Jinghao Jia, Michael V. Le, Salman Ahmed, Dan Williams, Hani Jamjoom, and Tianyin Xu. 2024. Fast (Trapless) Kernel Probes Everywhere. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 379–386. https://www.usenix.org/conference/atc24/presentation/jia
[14] Hsuan-Chi Kuo, Kai-Hsun Chen, Yicheng Lu, Dan Williams, Sibin Mohan, and Tianyin Xu. 2022. Verified programs can party: optimizing kernel extensions via post-verification merging. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 283–299. https://doi.org/10.1145/3492321.3519562
[15] Chang Liu, Byungchul Tak, and Long Wang. 2024. Understanding Performance of eBPF Maps. In *Proceedings of the ACM SIGCOMM 2024 Workshop on EBPF and Kernel Extensions (eBPF '24)*. Association for Computing Machinery, New York, NY, USA, 9–15. https://doi.org/10.1145/3672197.3673430
[16] Jinsong Mao, Hailun Ding, Juan Zhai, and Shiqing Ma. 2024. Merlin: Multi-tier Optimization of eBPF Code for Performance and Compactness. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 639–653. https://doi.org/10.1145/3620666.3651387
[17] Sebastiano Miano, Alireza Sanaee, Fulvio Risso, Gábor Rétvári, and Gianni Antichi. 2022. Domain specific run time optimization for software data planes. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 1148–1164. https://doi.org/10.1145/3503222.3507769
[18] Qiongwen Xu, Michael D. Wong, Tanvi Wagle, Srinivas Narayana, and Anirudh Sivaraman. 2021. Synthesizing safe and efficient kernel extensions for packet processing. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 50–64. https://doi.org/10.1145/3452296.3472929