Don't Stall Me Now: Hiding Memory Latency in eBPF

Farbod Shahinfar
Politecnico di Milano
Italy, Milan

Marco Molè
Politecnico di Milano
Italy, Milan

Aurojit Panda
New York University
United States, New York

Gianni Antichi
Politecnico di Milano
Italy, Milan
Queen Mary University of London
United Kingdom, London

Abstract

eBPF has emerged as a popular platform for building high-performance I/O programs. However, eBPF's programming model limits the use of commonly used performance optimizations, leaving programs vulnerable to cache-miss-induced performance overheads. Specifically, the programming model restricts the choice of data structures used by a program, thus limiting the programmers' ability to improve data locality. Furthermore, it also makes it hard for programmers to overlap computation with data movement.

In this paper, we present Beeswax an approach that addresses both of these restrictions. This approach requires programmers to adopt multi-phase data structures and partition programs into multiple stages. It overlaps the execution of program stages, and thus hides cache miss overheads. We designed the approach so that it could be applied to existing programs without requiring significant change. We have applied Beeswax to Katran, a load balancer deployed in production data centers, and BMC, a key-value store accelerator. We show throughput improvements of up to 99%.

CCS Concepts

• **Networks** → **Programmable networks**; **Network servers**; *Programming interfaces*; • **Software and its engineering** → Operating systems.

Keywords

eBPF, Software Prefetching, High Performance Network Applications

ACM Reference Format:
Farbod Shahinfar, Marco Molè, Aurojit Panda, and Gianni Antichi. 2026. Don't Stall Me Now: Hiding Memory Latency in eBPF. In *ACM SIGCOMM 2026 Conference (SIGCOMM '26), August 17–21, 2026, Denver, CO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3789240.3829175

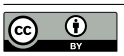
1 Introduction

Increasingly, high-performance I/O applications are implemented in eBPF and deployed in the Linux kernel's eBPF sandbox [18, 25, 28, 46, 59, 67, 70, 72–74]. This is because programs built in this manner have I/O overheads that are comparable to programs that use kernel bypass libraries, e.g., DPDK [22] and SPDK [62], but are easier to deploy and manage [65].

However, reducing I/O overheads is not sufficient for building high-performance applications. Even applications with minimal I/O costs, efficient processing logic and small memory footprints can have low throughput and high latency because of wasted processor cycles. For these applications, memory stalls caused by cache misses are a common cause of wasted processor cycles: when a program instruction tries to access data that is not in the processor cache (*a cache miss*), the processor pauses execution (*stalls*) until the data can be fetched into the cache. Because no processing is performed during the stall, application performance degrades. As we detail in §2.1, programmers rely on two techniques to mitigate the performance impact of stalls: improving cache locality through data structure design to reduce cache misses [15], and overlapping useful work with memory accesses by issuing prefetch instructions to asynchronously fetch data needed in the future [14]. Nearly all current high-performance I/O bypass programs rely on these approaches for performance [22, 62].

Unfortunately, as we detail in §2.2, applying either technique to eBPF programs is challenging. Despite recent improvements [64], adding new data structures to eBPF remains difficult. Consequently, most programs continue to use data structures provided by the runtime that were designed to be general (and usable across applications) rather than to maximize cache locality. Overlapping computation is even more complicated because of constraints imposed by the eBPF programming model: programs run-to-completion and must process an input in a fixed number of cycles. Consequently, eBPF programs perform limited computation per invocation and often have no processing that can be overlapped with asynchronous data fetches. The result is that, unlike in every other context, high-performance I/O programs written in eBPF do not currently mitigate overheads from cache misses. This is not just a theoretical concern; as we show in the next section, stalls due to cache misses reduce the performance of Katran [59], an L4 load balancer deployed in production, by up to 59%.

In this paper, we develop Beeswax, an approach that eBPF programs can use to mitigate stalls due to cache misses. The approach currently targets high-performance network I/O programs, e.g., network functions, but can be generalized and applied to other programs, including those performing I/O from storage devices. Our approach requires extensions to the kernel's eBPF runtime (§3)



SIGCOMM '26, Denver, CO, USA

ACM ISBN 979-8-4007-2467-1/2026/08
https://doi.org/10.1145/3789240.3829175

that enable the use of prefetch instructions from eBPF programs and batched processing so that programs can overlap computation for one packet with data prefetching for another. To apply Beeswax (§4), developers first perform syntactic analysis (and use profiling) to identify code locations that are likely to cause cache misses (§4.1). Guided by this set, they then replace data structures with application-tailored multi-phase data structures (§4.2) and partition program logic into multiple stages (§4.3). Beeswax provides a library to aid with implementing and using multi-phase data structures, and partitioning programs into multiple stages. At runtime, the library includes logic that interleaves stage execution across packets. This ensures that prefetching data required to process a packet is overlapped with the processing of a different packet, reducing the impact of cache misses.

We had three goals when designing Beeswax:

(i) It should not require changes to the logic in the kernel's eBPF runtime.
(ii) It should not add significant complexity to the (already complex) task of writing eBPF programs.
(iii) Applying it to existing eBPF programs should not require significant changes to program structure or logic.

These requirements also posed the main challenge in coming up with our approach, and we addressed these by carefully analyzing the eBPF programming model to identify where cache misses occur and how programs are structured. Given these observations, we identified a small set of changes that had to be applied to the eBPF runtime, thus meeting the first requirement. Our recipe for adopting the Beeswax approach (§4) is designed to meet the second two requirements.

We have applied Beeswax to two existing eBPF programs, Katran [59] and BMC [28], showing that our recipe allows Beeswax to be applied to existing, complex eBPF programs. Furthermore, our evaluation (§6) shows that applying Beeswax to Katran improves throughput by 18–99%. Our code, including changes to the Linux kernel, can be found at https://github.com/bpf-endeavor/bpf_prefetch.

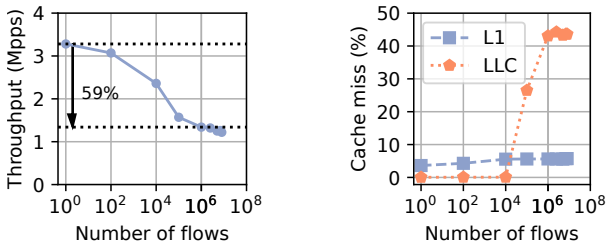
This work does not raise ethical concerns.

2 Motivation & Background

The performance of eBPF programs is influenced by CPU cache misses [40, 47, 57], which lead to memory stalls. We quantify the impact they have on eBPF based programs using Katran, an eBPF-based production-grade Layer-4 load balancer designed and developed by Meta [59].

We ran our evaluation by deploying Katran on a server equipped with an Intel Xeon Silver 4310 CPU with 1.1MB of L1 cache, 30MB of L2 cache, and 36 MB of last-level cache (LLC). A similar directly connected server was used as a load generator. We ran Katran on a single isolated core, allowing us to gather precise performance statistics.

Katran's working set size (and thus cache pressure) is proportional to the number of flows. Therefore, we measured throughput as we increased the number of flows to see the effect of cache misses. We found (Figure 1a) that throughput decreases by 59% as we increase number of flows from 1 to 1 million. For this test, the workload used a uniform flow distribution. We hypothesized



(a) (b)

Figure 1: Katran, a production-grade eBPF-based L4 load-balancer, is slower when handling larger number of flows (a); The reason is the increasing number of L1 and LLC cache misses (b).

this slowdown was due to cache misses: the only other possible cause would be collisions in the hash map that Katran uses for connection tracking, but across all our tests the hash map's load factor remains below 0.125 which makes collisions unlikely [55]. We confirmed this hypothesis by measuring cache misses per-packet, which increase around 1 million flows (Figure 1b).

But cache misses are not just a problem for Katran. Many common eBPF programs, including load balancers [50, 67, 70]; firewalls [44, 71]; proxies [9, 17, 39]; real-time communication [37, 49]; congestion control extensions [21, 29] and monitoring systems [16, 19, 61] maintain per-flow state. These programs also exhibit a data-dependent and irregular access pattern, and are thus susceptible to performance overheads from cache misses.

Unfortunately, standard approaches to ameliorating the effect of cache misses are not easily applied to eBPF programs. To better understand this limitation, we first review standard techniques commonly used to improve cache efficiency in software systems (§2.1) and then discuss the challenges of applying these techniques in the eBPF setting (§2.2).

2.1 Reducing Cache Miss Overheads

Modern processors already include hardware prefetchers that try to predict memory access patterns (such as sequential or stride accesses) and bring data from memory into cache to avoid stalls. However, this does not suffice for unpredictable accesses, for instance the ones affecting the Katran experiment above. Therefore, many high-performance programs combine two software techniques to reduce overheads due to cache misses:

Use data structures that improve cache locality. Prior work [15, 27, 48, 55] has shown that careful data structure layout can improve cache locality and reduce the chances of misses. Layout changes including removing indirection, clustering (into the same cache line) data likely to be accessed at the same time, and reducing data size can all reduce cache misses, and thus avoid overheads.

Overlap other computation with data movement. Some cache misses are inevitable, and cannot be solved just by careful data layout. Programs reduce the effect of these stalls by using prefetch instructions (provided by nearly all modern CPUs) to asynchronously fetch data from memory into caches. The prefetch instructions returns before memory has been fetched, and the programs can use this time to run other computation [5, 14, 41]. A significant

challenge when using this approach lies in identifying what computation should be overlapped, and many high-performance I/O programs (e.g., kernel bypass network functions) process batches of inputs to enable prefetching.

2.2 Barriers to Reducing Cache Misses in eBPF

Applying either technique to eBPF programs is non-trivial because of eBPF's limited programming model. The programming model limits the program's ability to control data layout, asynchronously fetch data, or interleave computation. The net result is that current eBPF programs do not use either technique. We elaborate on the barriers imposed by the programming model below.

Custom data structures. eBPF programs expose a restricted programming interface [45, 58]. Programs use pre-allocated, kernel-managed data structures accessed through a fixed API (e.g., lookup and update operations) to persist state across invocations. These data structures, which are collectively referred to as *maps* [26], are intentionally opaque and do not expose memory layout or algorithmic details. This simplifies programming and allows Linux kernel developers to change implementation without breaking compatibility. But this opacity means that developers have nearly no control over data placement or cache locality.

Recent extensions, in particular eBPF Arena [64], partially relax these constraints. They provide eBPF programs with a page allocation API. The addition of a Arena makes it feasible to develop application specific memory layout and implement data structures in eBPF, but this is more challenging than implementing data structures from scratch in other contexts because the eBPF verifier limits the number of instructions that each function can run. This constraint also makes it hard to port existing libraries to the Arena API. Consequently, cache efficient structures have not yet been adopted by eBPF programs.

Overlapping computation. The barriers to overlapping computation are higher: the eBPF runtime assumes a strict run-to-completion model, where a single input (a packet, block, etc.) must be processed completely. Therefore, on each invocation a program has a limited amount of computation to perform and cannot yield control to a different program. As a result, there is no computation it can overlap with an asynchronous memory fetch. Worse, the eBPF runtime does not currently allow programs to issue prefetch instructions. Consequently, eBPF programs today *cannot* overlap computation to avoid cache miss overheads.

Our approach, Beeswax, which we describe next aims to remove these barriers and thus allow eBPF programs to avoid cache miss overheads.

3 Beeswax Overview

Beeswax's approach to ameliorating the effects of cache misses relies on two core techniques:

(a) It changes eBPF programs, so that instead of processing a single packet at a time, they process batches of packets[1]. This change allows the program to overlap data loads from memory into cache (triggered using prefetch instructions) with the processing of other packets.

(b) It enables the use of *application-tailored multi-phase* data structures. The use of application tailored-data structures improves access locality, increasing the chance that frequently accessed data remains in cache, while the multi-phase API provided by these data structures allows prefetch instructions to be used even with complex data structures like trees.

[1]As a reminder, our focus in this paper is on eBPF based network programs, and thus we focus on packets here. But batching can also be effectively used in storage or other contexts as demonstrated by SPDK [62] and other high-performance I/O libraries

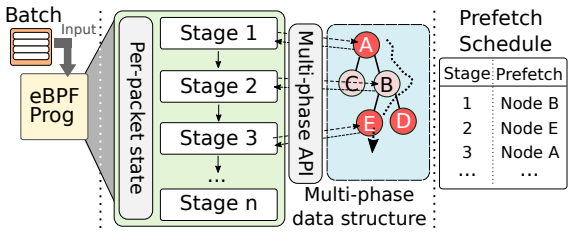


Figure 2: An example Beeswax program performing LPM based routing. Packets are processed in batches by an eBPF program structured as a multi-stage data-flow graph, where stages incrementally process packets, prefetch data for subsequent stages, and use shared memory to track per-packet state. Beeswax implements data structures in an explicitly accessible memory region, rather than opaque eBPF maps, and exposes an API for interleaving data structure operations with packet processing.

Enabling this approach required us to (i) extend the eBPF runtime to enable batching and prefetching (§3.1); and (ii) develop programming recipes that can be used to build data structures and programs (§4). To simplify the adoption of this approach, Beeswax provides a library that provides syntactic support and common abstractions that reduce programmer effort when adopting our approach.

3.1 Runtime Extensions

We made two changes to the Linux kernel's eBPF runtime:

(1) We added helper functions that the eBPF program can use to issue memory prefetch instructions. This change allows the program to trigger data loads from memory into cache before access, thus avoiding memory stalls.

(2) We changed eBPF hooks [4] so that they provide batches of packets, rather than individual packets. This change ensures that the eBPF program has other computation that it can perform while data is loaded into cache.

In addition, we use the Arena [64] APIs to build *application-tailored multi-phase data structures*. These data structures have better data locality than eBPF native data structures, and explicitly support prefetching via a *multi-phase* access API. Arean is a recently introduced feature that allow eBPF programs to allocate and directly access memory pages, however the Arena feature itself does not implement any data structures on top of this allocated memory.

3.2 How Beeswax Helps

We now describe how programs can use these runtime features to reduce memory stall overheads. Our changes alter what data structures the program uses, and how the program logic processes inputs. We illustrate our description using a longest-prefix match based routing program, which we illustrate in Figure 2. We also use

the same program as a running example throughout the paper, and also present performance results for it in §6.2.

Data structures. First, Beeswax programs use application-tailored data structures, built using eBPF Arenas to store data, e.g., routing tables. These structures are stored in memory allocated and managed using its API [64]. As we discuss in §4.1, the use of an appropriately designed application-tailored data structure can improve access locality, and thus reduce cache misses (and stalls), in cases where most accesses are to hot items.

In addition, Beeswax programs use data structures that offer multi-phase APIs to reduce the effect of cache misses for other access patterns: a single data structure operation (e.g., a `get` from a hash map stored as a tree) is split into multiple phases, each of which returns control to the caller when an access is likely to cause a cache miss. This allows the caller to perform computation while the data required by the data structure operation is fetched from memory into cache. In our running example, the program uses a trie[2] to implement longest prefix match (LPM) search. The multi-phase API for this trie processes one level at a time, and issues a prefetch instruction for the next level before returning control to the program. For example, in the Figure 2, the first stage accesses node `A`, issues a prefetch for node `B`, and returns control to the program. The program can then interleave computation for a different packet while the data is prefetched.

Program logic. Second, Beeswax programs operate on *batches of packets* and are split into *stages*. As mentioned above, we needed to change the eBPF runtime to enable batched processing: we had to change hooks so that they provided packet batches as input, but processing a batch requires more instructions. The eBPF verifier limits the number of instructions executed by a call, and thus to support batching we increased this limit to support this. In addition, the Beeswax library (described below) also uses tail calls [11] and global functions [63] to avoid interfering with the verifier.

To aid in this process, the Beeswax library provides syntactic support to split the program's processing logic into *stages*. Each stage (e.g., Stage 1, also see Lines 13–19 in Snippet 1) is written to process one packet at a time. At runtime, stages are executed sequentially, i.e., the first stage first processes all packets in the batch, then the second stage processes all packets, and so on. This model allows data fetches from memory into cache for one packet to be overlapped with the processing of another: for example, when processing packet `0` in the batch, Stage 1 can use the prefetch helper function (which Beeswax adds) to load data that Stage 2 will require when processing packet `0`. While the processor is fetching the data, stage 1 processing for other packets in the batch (e.g., packet 1) is run, thus ensuring that processing cycles are not wasted. The Beeswax library also provides additional abstractions, including a per-packet state structure that is shared across stages, to reduce the effort required to partition program logic into stages and so that the program stack size does not need to change proportional to batch size.

[2]Trie is a tree structure which encodes keys in the path from the root to nodes.

4 Applying Beeswax

In this section, we describe how a programmer can apply the Beeswax approach to an eBPF program. First, the programmer must identify accesses that are likely to cause a cache miss (§4.1). Next, the programmer needs to restructure logic so that they use prefetch instructions to load data into memory, and overlap other computation to use the CPU resources. What this restructuring looks like depends on whether the miss happens in a data structure operation (§4.2) or in program logic (§4.3). We describe both, and then in §4.4, we provide an example of applying Beeswax to an LPM router.

4.1 Identifying Likely Cache Misses

Care must be taken when using prefetch instructions: indiscriminate use increases code size and thus pressure on the i-cache, can interfere with the hardware prefetcher [36], and consume CPU cycles without benefit if the data is already in the cache. Thus, to apply Beeswax, a developer must identify accesses that are likely to cause a cache miss. We provide a two step approach to do so: the programmer starts by analyzing the program source code to identify a likely set of places where cache misses could occur; and then uses a profiler to refine the identified set and ensure its validity.

4.1.1 Source Code Analysis. We derived our source code analysis approach by examining the access patterns of eBPF programs. Most such programs access data from three locations: (a) the stack, (b) eBPF maps, and (c) the packet data being processed. The likelihood of a cache miss depends on the location being accessed, as discussed below.

Stack. eBPF programs use the stack for local variables that do not fit in registers (and thus spill over). As is true for other programming environments, programs exhibit strong temporal locality when accessing stack variables, and thus access to stack variable are unlikely to result in cache misses. Consequently, we do not add these accesses to our set of likely-cache-miss locations.

Maps. eBPF programs use maps for a variety of purposes, including to store state between invocations (i.e., as a global variable), for configuration, etc. Programs access maps by calling lookup (`bpf_map_lookup_elem`) to get a pointer to the value, and then dereference the returned pointer. Cache misses can occur in the lookup logic, or when the pointer it returns is dereferenced. For both places, the likelihood of a cache miss depends on the program's access pattern, and we need to consider three cases:

(1) The value could be a global variable[3], or an index that is frequently accessed (e.g., a histogram bucket).
(2) Sequential memory access (e.g., array iteration).
(3) Random memory access (e.g., traversing trees).

Cache misses are unlikely to be a problem for the first two access patterns: Frequently accessed data remain in cache due to access locality, especially for packet processing programs engineered for high-performance (which tend to have small working set sizes). In contrast, for sequential memory accesses, the hardware prefetcher [30] limits processing overheads.

[3]In eBPF global variables are stored as an ARRAY map with single entry.

```
1  // Initialize the search
2  finished = 0;
3  BAX_STAGE(FIRST, {
4    ...
5    pstate->key = *r;
6    lpm_phase1(dat, &pstate->partial_state);
7    BAX_NEXT_STAGE(DO_LOOKUP);
8  })
9  // Walk the trie for packets in the batch
10 for (int i = 0; i < DAT_KEY_SIZE_BIT; i++) {
11   if (finished >= batch_size) break;
12
13   BAX_STAGE(DO_LOOKUP, {
14     int ret = lpm_phase2(dat, &pstate->key, 32,
15                          &pstate->partial_state);
16     if (ret) continue; // Continue to next phase
17     finished++; // Count of operations finished
18     BAX_NEXT_STAGE(PREPARE_PKT);
19   })
20 }
```

Snippet 1: A Beeswax program using multi-phase API to overlap multiple LPM operations.

```
1 // Type of per-packet state
2 typedef struct {
3     // Required: next stage for the packet.
4     int stage;
5     int key;
6     // partial lookup state defined by LPM library
7     struct dat_state partial_state;
8     // ...
9 } pkt_state_t;
```

Snippet 2: Declaration of per-packet state used in Snippet 1.

Thus, the programmer only needs to focus on random accesses, issuing prefetch instructions to avoid misses both in the lookup logic and when dereferencing pointers. Therefore, we add both to the set of likely-cache-miss locations.

Packet. Modern processors include features (e.g., Intel's DDIO or AMD's SDCI) that load first 64-bytes (cache line) of each packet into cache. However, an access to a different cache line can induce a miss. Thus, we add any accesses outside the first 64-bytes of a packet to the set of likely-cache-miss locations.

4.1.2 Profiling. Once the programmer has identified a likely-cache-miss location set from the source code, they next run the program using a representative workload and use a profiler to observe cache misses. Profiling helps reduce the size of the likely-cache-miss location set. This is because workload characteristics and differences between processor architecture [31] can result in no cache misses at some of the locations.

4.2 Designing Data Structures

Beeswax applications use custom data structures for two reasons: to have greater control over data-layout, thus improving access locality, and to allow programs to hide the effects of cache misses within the lookup logic when accessing random elements. Designing data structures that have better locality has been studied extensively [2, 55], and we merely adopt these approaches. Therefore, here we focus on data structure interfaces that allow computation to be overlapped with data fetches from memory into cache.

Observe that the interface needs to balance two opposing concerns: it needs to be such that an application programmer needs limited visibility into data structure internals, while at the same time allowing the application programmer to specify what computation is overlapped with a prefetch instruction issue by the data structure. In order to meet these competing requirements, Beeswax adopts a multi-phase interface: a single data structure operation (e.g., lookup in a LPM trie) might be split into several phases (e.g., in Snippet 1, lookup is split into two phases one on line 6 and the next on line 14). A multi-phase interface allows the data structure to issue prefetch instructions within a phase, return control to the program which runs other computation, and then access the fetched data in a different phase.

Of course, when adopting this approach a data structure programmer must first identify phase boundaries. The likely-cache-miss location set identified above helps with this: it provides a set of program locations before which prefetch instructions should be issued, and the programmer simply creates a stage before these locations. Note, that care must also be taken when using data structures with multi-phase APIs to ensure that the overlapped computation does not evict the prefetched data.

Finally, implementing these data structures presents an additional concern. When applying Beeswax, we implemented data structures in eBPF and used the Arena APIs to manage memory. Beeswax applies to alternate approaches for building data structures, including using Kflex [24] or implementing it in kernel modules [69].

4.3 Designing Program Logic

Beeswax programs need to be structured so that computation can be overlapped with data fetches from memory that are triggered by prefetch instructions. For instance, when using a multi-phase data structure, the program must interleave other computation between phases. The same is true when the program issues a prefetch instruction because accessing data. Thus, applying Beeswax to a program requires identifying what computation should be overlapped with what prefetch instruction.

To aid with this, Beeswax provides syntactic support for partitioning the program into *stages*. As shown in Snippet 1, each stage (lines 3–8 and 13–19) is written as if it operates on a single packet and can specify the next stage that should be invoked (line 7 and 18). In most cases, a stage boundary represents the place where further computation requires accessing data that is being fetched from memory, and thus overlapping other execution is helpful. For example, in the `FIRST` stage, processing the packet requires access to the appropriate first level of the LPM trie, but this needs to be fetched from memory into cache. Thus, the stage boundary indicates that other processing should be overlapped here.

At execution time, Beeswax repeatedly iterates over the batch of packets, applying the appropriate stage to each packet. For example, initially the `FIRST` stage processes all packets in the batch, and thus packet 0's memory fetch is probably overlapped (the exact set of packets that are processed depends on the time taken by the prefetch and by the processor) with the `FIRST` stage executing for

Keyword	Description
`batch_size`	Number of packets in the batch
`pstate`	Pointer to state of current packet
`pkt`	Pointer to original XDP context
`data, data_end`	beginning and end of packet
`BAX_k`	index of current packet in the batch
`BAX_BEGIN_OF_FILE()`	Mark beginning of source code file
`BAX_END_OF_FILE()`	Mark end of source code file
`BAX_DECLARE_INIT_STAGE_NAME(name)`	Name of the first stage
`BAX_DECLARE_PKT_STATE_TYPE(type)`	Declare the data type of the per-packet state
`BAX_PROG_BEGIN()`	Mark beginning of a program function
`BAX_BREAK_TAIL_CALL()`	Mark a breaking point. The program will be split into two programs chained with tail call
`BAX_INIT_BATCH_STATE()`	Initialize the batch state
`BAX_STAGE(name, {...})`	Define a stage with the given name
`BAX_NEXT_STAGE(name)`	Set the next stage for the packet
`DROP(),PASS(),TX()`	Set the verdict for the packet
`ACTION(value)`	Set the verdict for the packet based on value
`BAX_GET_PSTATE(`k`)`	Get the k^{th} packets state
`P(addr)`	Prefetch the memory address

Table 1: The Beeswax library.

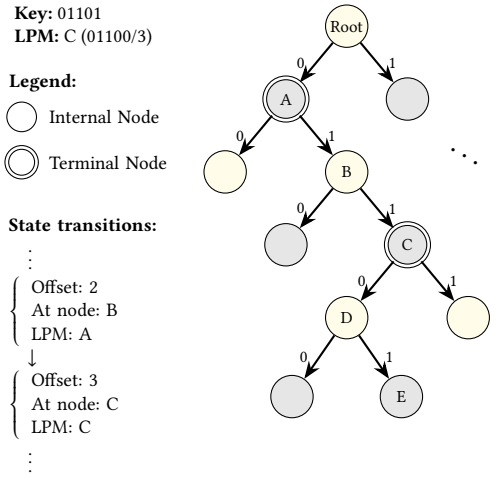


Figure 3: An example of how each invocation of multi-phase API progresses through the trie data structure to find the LPM.

packet 1, etc. The Beeswax library provides additional syntax to help partition programs (Table 1).

Maintaining Per-packet State. The stage syntax provided by Beeswax allows programmers to specify how individual packets are processed. In many cases, the program might need to pass state from one stage to another, and to aid with this the Beeswax library provides a per-packet state abstraction. The abstraction is implemented using a per-CPU array [3]. Each element of the per-CPU array corresponds to a batch, and is itself an array containing state for each packet. We chose this design because prior work [46] has found that this performs better than other approaches.

Syntactically, programmers define a structure (Snippet 2) that specifies the per-packet state that should be maintained and then access this state using a `pstate` variable within each stage (e.g., line 6 of Snippet 1). We require that the per-packet state structure contain a stage index, and at runtime Beeswax uses this to track what stage should be used to next process a packet.

4.4 An Example: LPM In Action

Finally, we show how Beeswax is applied to an LPM router, such as the one shown in Figure 2. As a reminder, we are considering a case where LPM lookups are performed using a multi-phase trie. The program logic is shown in Snippet 1.

The first stage (lines 3–8) extracts the lookup key, initializes the per-packet state required for the LPM operation, and issues a prefetch for the root of the trie, the first node accessed during traversal. The program then marks packets for transition to the next stage. This stage transition provides the time window needed to hide the latency of fetching the root node into the cache.

Lines 13–19 define an iterative stage that drives the multi-phase LPM operation concurrently for all packets in the batch. Each invocation of the API compares a bit of the key and advances the traversal by one level in the trie (line 14). Figure 3 illustrates this process. For example, consider the key `01101` and assume that two levels have already been traversed, with the current node at *B*. The next API invocation advances the traversal to node *C*; since *C* is a terminal node, the LPM result is updated accordingly.

5 Implementation

Implementing Beeswax required changing the Linux kernel and implementing logic outside. The kernel changes add 1700 lines of C code to the Linux kernel. We modified version 6.15.0. The changes we made included:

(1) Changes to the Mellanox and virtio drivers: support for batching.
(2) Changes to the eBPF runtime and XDP hooks: support for batching and runtime extension.
(3) Changes to the eBPF verifier to increase instruction limits.
(4) Adding an eBPF helper to add a prefetch instructions, and changes to the eBPF JIT to inline calls to this helper.

The drivers were changed to call a new batched XDP hook. Both drivers already processes packet batches, and we changed the the driver to invoke the hook with these batches. No changes were required to the batching logic.

Runtime changes were required for two reasons: (a) to make the verifier aware of the batched XDP hook and increase complexity limit proportional to the batch size; and (b) to inline calls to the prefetching helper, thus reducing function call overheads.

In addition, the Beeswax library consists of 383 lines of AWK and C code. This is used to translate multi-stage program logic written using the Beeswax APIs (Table 1) into C code that can be compiled to eBPF.

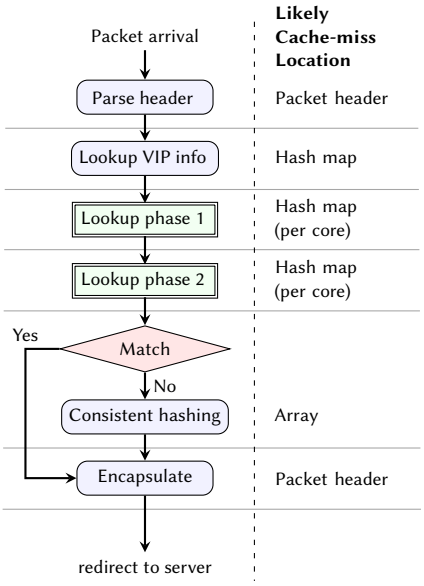


Figure 4: Flowchart of Katran's packet-processing pipeline. The horizontal lines show how we divided the program into stages to run it with Beeswax. The green boxes show the phases of the hash map used for connection tracking.

6 Evaluation

Our evaluation focuses on two questions:

(1) By how much does performance improve when Beeswax is applied to real-world applications (§6.1)?
(2) What is the performance impact of each of the design choices, including multi-stage program structure, batched execution and application-tailored multi-phase data structures, made by Beeswax (§6.2)?

Baselines. We answer the first question, by applying Beeswax to Katran, a production-grade Layer-4 load balancer used by Meta, and to BMC [28], an academic system that relies on eBPF to accelerate Memcached workloads.

For the second question, we use a set of microbenchmarks based on two applications with distinct functionality and performance characteristics. The first microbenchmark represents a forwarding workload and implements an LPM-based router. The second represents a monitoring workload and is based on the CVM algorithm [12, 35], which estimates the number of distinct IP addresses observed in a packet stream.

Experiment Setup. All experiments were run on Cloudlab [23] `c6525-25g` servers. Each server is equipped with an AMD EPYC 7302P 16-Core (Zen 2 – Rome) processor with 0.5 MB of L1, 8 MB of L2, and 128 MB of LLC; has 128 GB of memory; and a Mellanox Connect-X5 (MT27800 Family) NICs. We used two directly connected servers for each experiment: one served as the load generator while the other ran the Beeswax program. Both servers ran Ubuntu 22.04, but the server running Beeswax used a modified version of the kernel with support for Beeswax. During experiments the eBPF or Beeswax programs are allocated one core.

6.1 Beeswax Applied to Applications

Katran. We used the approach from §4.1 to identify possible miss locations in Katran [59], and identified that reads and writes from the hash table that tracks active connections were a likely source. Profiling using `perf` confirmed that this was the case.

Given this, we applied Beeswax by first creating a two-phase version of the eBPF hash map (`BPF_MAP_TYPE_HASH`, a chained hash map): given a lookup key, the first phase identifies the correct map bucket and issues a prefetch to load it into cache; while the second phase traverses the bucket's list to find the correct entry. Figure 4 illustrates the resulting program logic. Dashed lines in the figure indicate stage boundaries: we can see that each phase of the LPM lookup runs in its own stage. *These changes added 720 lines of code to Katran.*

In Figure 5 we evaluate the efficacy of the Beeswax approach by comparing baseline Katran's throughput to the version with Beeswax applied. We evaluate performance as we increase the number of flows, which affects the size of the hash table and thus the program's working set size. For our workloads, we use a Zipfian flow distribution, and vary the parameter across runs. Workload skew affects the likelihood that a packet belongs to a 'popular' flow whose state is in cache. Our results show that Beeswax improves throughput across these settings, and improves throughput by 18–99% compared to the baseline. As expected, Beeswax has greater effect when there are many flows and the workload has lower skew, as this increases the chance of a cache miss when processing a packet.

BMC. Applying our approach to find likely-miss-locations to BMC [28] yielded similar results as Katran: misses were most likely to occur when a hash map was used. There are two important differences: the hash map in this case caches Memcached keys-and-values (rather than virtual IP addresses), and the hash map is an open-address hash map.

Our approach to applying Beeswax was also similar: we added a multi-phase interface to the hash map: the first phase finds the index of the key and issues a prefetch instruction for the value, and the second phase reads and returns the value. We split the program logic into stages to overlap these phases for different requests. *These changes added 407 lines of code to BMC.*

We evaluated the resulting performance using a workload derived from Facebook's Memcached workload [6]. We use Mutilate [1] to first load records, and to then issue UDP requests[4]. Both value size and request distributions are derived from the Facebook workload.

Figure 6 compares throughput as we vary the number of records stored. In this experiment, the number of records controls working set size, and thus cache misses are more likely with larger number of records. We observe that with a million records, Beeswax improves throughput by 17%. However, we also observe a 3% drop in throughput when the store contains a single record. This is because in scenarios where there are a small number of records, they are likely to remain in cache. In this case, the multiple API calls and program stages required by Beeswax add overhead without providing any benefit, leading to the decrease. We found that Beeswax

[4]We use Memcached ASCII protocol as required by BMC.

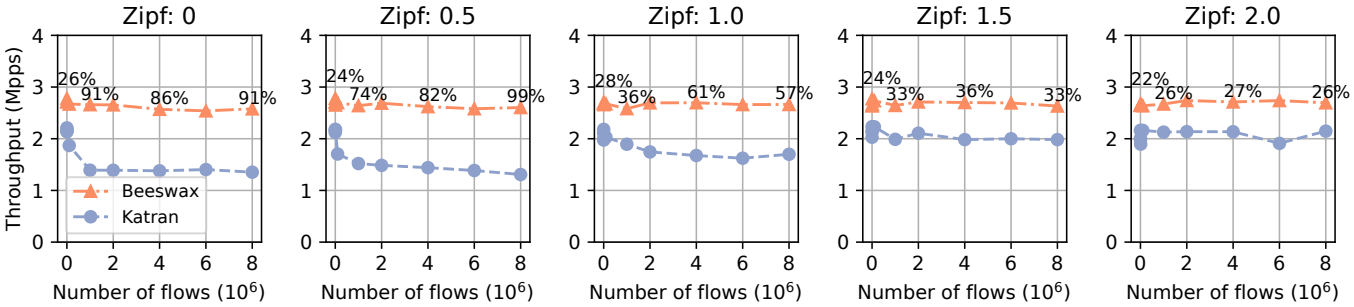


Figure 5: Evaluating the benefit of using Beeswax for Katran, across different number of flows (a proxy for working set size) and different workload skewness. We see that by increasing the number of flows Katran experiences more cache misses and throughput drops while Beeswax can hide this effect and maintain throughput. More skewed workloads (higher values of Zipfian parameter) put less pressure on the cache so the effect of increasing the number of flows are less bold. In either case, Beeswax is maintaining a higher throughput. (Numbers above the line shows percentage of improvement after applying Beeswax).

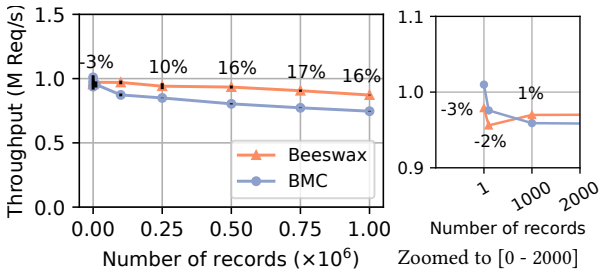


Figure 6: Comparing the throughput of BMC and its Beeswax version using a real-world workload from Facebook [6]. Applying Beeswax to BMC improves throughput by up to 18.4% when the program is experiencing cache misses. When only one record is used and there are no cache misses, the overhead imposed by Beeswax multi-stage design is visible.

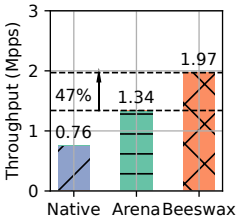


Figure 7: Performance of an LPM router with three LPM implementations: Native eBPF map, in Arena and in Beeswax.

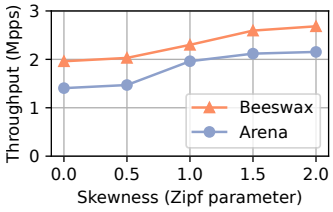


Figure 8: Throughput comparison between Beeswax and an Arena implementation for LPM-based router as traffic skew increases.

breaks even (i.e., has better performance) when a 1000 records are used.

Takeaway. These two cases show that applying Beeswax to applications can provide substantial throughput improvements when cache misses are a significant source of slowdown: for Katran we improved throughput by up to 99%. However, in cases where cache misses have no effect, the overheads introduced by splitting program logic into stages and data structure APIs into phases can hurt performance.

6.2 Microbenchmark

This section first analyzes the performance implications of data structure design (§6.2.1), and then evaluates how program structure and execution influence performance (§6.2.2).

6.2.1 The Impact of Data structure Design. Our first set of microbenchmarks evaluate the benefits of tailoring data structures to applications, and adopting a multi-phase API. Our evaluation in this case is performed using a router that looks up a packet's destination IP in a tree-based LPM trie, which has been our running example throughout the paper.

Does applying Beeswax to a data structure improve performance? We implemented the Beeswax LPM trie in eBPF using the Arena APIs to manage memory. First, in Figure 7 we quantify the performance difference between the native eBPF LPM structure (Native), an implementation using the Arena API but without prefetching or a multi-phase interface (Arena), and a Beeswax implementation. We compared the performance of all three approaches by instantiating the router with 980 thousand prefixes from an IXP rule set[5], and choosing addresses uniformly from within the valid range. Our results show that (a) *efficient data structures can be implemented in eBPF*: both Arena implementations perform better than the native implementation, which we hypothesize is because calls to the lookup function are cheaper; and (b) *the Beeswax approach provides significant performance improvements*: the Beeswax implementation has 47% better performance than the eBPF Arena version. Finally, in Figure 8, we demonstrate that these conclusions generalize across traffic patterns by using a Zipfian distribution to pick among destination addresses, and varying the skew factor. We observe that across all skew factors, Beeswax outperforms the Arena implementation.

Does tailoring data-structures to applications help with performance? Next, we evaluate the benefit of tailoring data structures to applications by comparing two implementation of the LPM trie. Both implementations use the Beeswax approach, i.e., have multi-phase APIs and use prefetching. They only differ in data layout: one

[5]Namex IX, November 2024 from RouteViews [51] archive. File `rib.20241130.2200` from https://archive.routeviews.org/namex.fco/bgpdata/2024.11/RIBS/

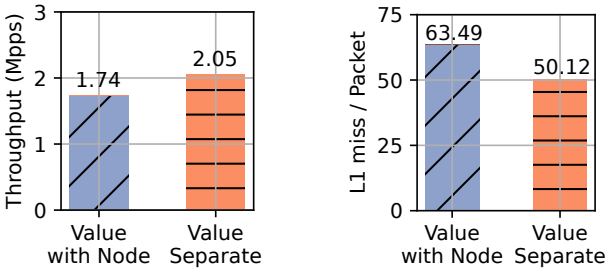


Figure 9: The benefit of tailoring data structures to the application: value with node has better spatial locality (values and nodes are co-located) but performs worse for our application because of increased cache pressure.

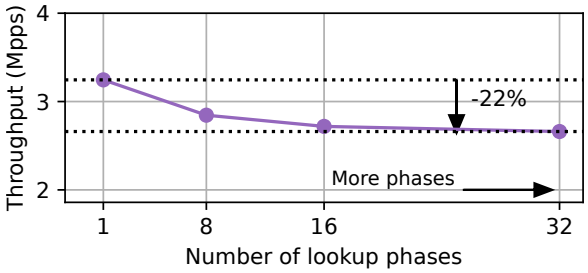


Figure 10: In an LPM-based router with only one rule, the working set fits in cache, and the multi-phase data structure incurs overhead without providing benefit.

(value with node) stores the value in the trie node, and the other (value separate) stores values in a different location.

While storing values within the node increase locality when accessing the value (it is likely to be in the same cache line as the terminal node) they also increase cache pressure when traversing the tree (because each node is larger). In our setup, the effects of the increased cache pressure dominate: Figure 9a shows that putting the value in a different node improves throughput by 17.8%. Figure 9b shows L1 cache miss rates for both designs, demonstrating that this is because of lower cache pressure.

However, an alternate application where tree has smaller height (and thus fewer nodes are visited during each lookup) or where values are smaller, might show better performance when the value is in the node. Indeed, the native eBPF implementation stores values in the same node.

How many phases should a data-structure use? Splitting an API into phases has benefits (it allows computation to overlapped with data fetches) but also comes at a cost because it increases the number of function calls made by the program. We measure the overhead of additional stages by instantiating the LPM router, and configuring it with one rule. This ensures that all nodes used during lookup remain in cache, and thus (similar to our observations with BMC) makes the use of multiple phases or prefetching unnecessary. We then vary the number of API phases and measure throughput. Our results, in Figure 10 show that as the number of phases increases throughput drops: when using 32 phases (each phase accesses a single trie node) the router has 22% lower throughput than the 1 phase version. This result demonstrates the importance of using a

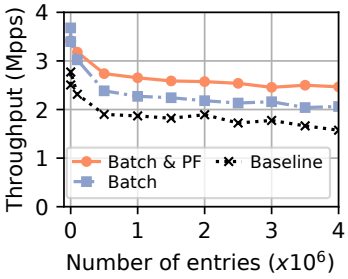


Figure 11: Using batching and prefetching can effectively hide cache miss stalls and improve performance up to 46.87%.

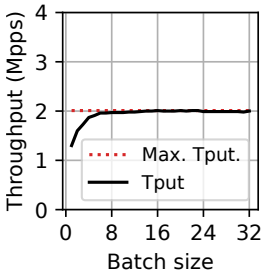


Figure 12: Batch size impact on throughput performance using the native eBPF LPM structure.

profiler to confirm the set of likely-cache-miss locations identified by syntactic analysis (§4.1.2) to avoid unnecessary overheads from additional phases (or program stages).

6.2.2 The Impact of Program Design and Execution. Our final set of microbenchmarks measure the effect that batching and batch size have on performance. The previous microbenchmark on data structure phases also applies to program logic stages, and we thus do not evaluate it here.

How does batching affect performance? How important is prefetching? First, we use the LPM router to investigate the performance impact of batching and prefetching. In Figure 11, we measure the performance of a version that uses no batching (Baseline), a version that uses batching but does not prefetch data (and whose processing logic is not split into stages), and a version with Beeswax applied that uses both batching and prefetching. We used a batch size of 8 for the latter two versions. We observe that batching without prefetching improves performance when compared to the baseline by 47%. This is because batching reduces overheads from calling an eBPF hook [57]. Prefetching improves performance by a further 23.3% (when compared to just batching) by reducing memory stalls. Thus, we conclude that both batching and prefetching when used together provide significant performance improvements.

What batch size should be used? As we saw, batch sizes affect performance: a small batch size reduces the amount of computation that can be overlapped, while a large batch size increases the chance that data fetched into cache is evicted before it is used. In Figure 12 we explore how the LPM router's throughput varies as we change batch sizes. We observe that a batch size of 8 yields the highest throughput, and smaller batch sizes have significantly lower performance.

But is batch size 8 appropriate for all applications? We investigate this by implementing another eBPF program that uses the CVM [12, 35] algorithm to count the number of distinct destination IP addresses received. This program uses a randomized treap [56].

We first checked that a batch size of 8 continued to be effective for this program: Figure 13 compares the throughput achieved of a baseline version (without batching or prefetching) to a Beeswax version. For this experiment we used a workload where destination IP addresses were drawn uniformly at random. Our results show

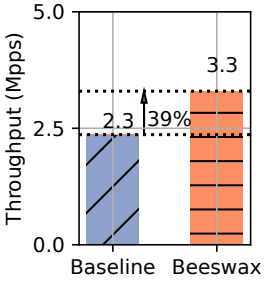


Figure 13: Comparing throughput of a naive implementation of CVM with Beeswax version.

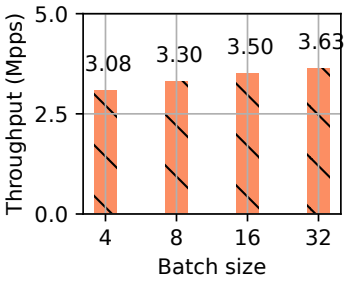


Figure 14: Throughput of Beeswax implementation of CVM application with different batch sizes.

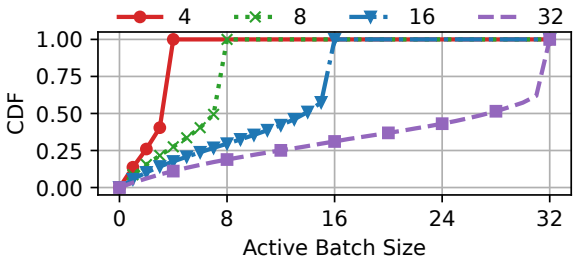


Figure 15: Distribution of active packets in each stage when using Beeswax version of CVM.

that a batch size of 8 is effective: the Beeswax version has 39% higher throughput.

Next, we varied batch size and measured how it affected throughput for the CVM program. Our results, in Figure 14, reveal a 10% increase in throughput when batch size increases from 8 (3.3 Mpps) to 32 (3.63 Mpps). This indicates that the appropriate batch size depends on the application.

Finally, we wanted to understand why larger batches improved throughput for the CVM program. We hypothesized that this might be because of variance in the length of the treap path traversed by each packet. If this were true, when small batches are used, later stages of the program would not be able to overlap computation (because there is no computation to run) with prefetch instructions, and would thus see a memory stall. We tested this hypothesis by instrumenting the program. Figure 15 illustrates the output of our instrumentation: we plot a CDF of the fraction of packet available to process at any stage. Indeed, the distribution confirms our hypothesis. Thus, program logic has significant impact on the choice of batch size for Beeswax applications.

7 Related Works

Related work on eBPF can be organized along three dimensions: techniques for improving eBPF performance, approaches that enable application-specific data structures, and methods for identifying memory accesses that are prone to cache misses. We discuss these, and then discuss the work beyond eBPF.

Optimizing eBPF program performance. X2DP [52] is the most closely related work, and it explores processing batches in eBPF, which enables prefetching and the use of SIMD instructions to achieve higher throughput. However, unlike our work, X2DP depends on a programmable NIC to produce batches, and does not explore approaches to avoid cache misses when accessing data structures.

Other work has explored compiler extensions to improve eBPF program performance. These proposals include switching to superoptimization based compilers [68], requiring new compiler optimization passes [43], using a cost-model and planner optimization phase inspired by SQL databases [60], and using profile-guided optimization [47]. First, Beeswax is complementary to these efforts and can benefit from their optimizations. Second, these optimizations suffer from architectural blindness [57], which Beeswax specifically addressed for prefetch instruction, a more thorough but similar design can extend these compilers to further optimize eBPF programs.

Improving eBPF data structures. Several recent works have developed more efficient data structures for eBPF programs. Of these, BPF-DB [10] and Kerby [34] are implemented using memory allocated from an eBPF map, and eNetSTL [69] is implemented as a kernel and eBPF runtime extension. These specialized data structures can improve cache locality, thus reducing number of cache misses. However, they do not provide a multi-phase API, and thus cannot entirely reduce cycles wasted due to cache misses.

Recently, Rex [32] and KFlex [24] have proposed eBPF runtime extensions that remove some programming model constraints by either eliminating the verifier (Rex) or weakening the invariants enforced by the verifier (KFlex). This makes it easier to implement complex data structures in eBPF. However, neither address the issue of cache misses, and are complimentary to our efforts: they might reduce the effort when applying Beeswax to a program.

7.1 Beyond eBPF

Identifying likely-cache-miss locations. Identifying likely-cache-miss locations is an important problem for several domains including real-time computing. Several static approaches have been proposed for this, the majority of which focus on computing worse-case execution time (WCET) [42]. Recently, CFAR [31] has proposed an alternative approach that provides more flexibility in reasoning about cache accesses. Other work that targets databases [13], has designed approaches to predict access and hide misses for hash-join operations. Dynamic techniques, where profilers, e.g., Perf [20], are used to identify cache misses are much more commonly used. Tools such as InXpect [53] specialize these tools to the XDP and eBPF context. Our approach to identifying likely-cache-miss locations (§4.1) can be augmented with any of these static or dynamic tools.

Non-eBPF Packet Processing Systems. Prior work has also described approaches to avoid misses in packet processing pipelines, e.g., NF chains and software switches. Fajita [27] is an extension to FastClick [7] that aggregates multiple hash maps into one, thus improving locality and enabling better prefetching. ESwitch [48] applies a similar approach to OvS. However, applying either to eBPF is challenging because of the additional complexity they add. G-Opt [33], proposes a more general approach 'group prefetching' that aims to hide memory access latency for packet processing

applications by splitting processing into phases, and prefetching data. The resulting programs resemble our multi-stage approach.

Batchy [38] shows that batching can be detrimental in the presence of divergent control flow, and develops an approach to address this problem. Figure 14 and 15 in our figure show that this is a problem for eBPF too, however, Batchy's solution cannot directly be applied to eBPF, and alternates are necessary.

Alternate hardware interfaces. ENSO [54] and ASNI [66], which propose new hardware interfaces for NICs are orthogonal to our work, and Beeswax can be used with these.

8 Discussion

Finally, before concluding we discuss how Beeswax can be generalized beyond network programs, and its impact on the eBPF ecosystem.

Beyond network programs. In this paper, we focused on packet processing programs written in eBPF. Prior work [8, 72] has also discussed using eBPF for other I/O heavy programs, including those that work on file systems. These programs also have unpredictable access patterns to traverse complex data structures (e.g., B-trees), and can also benefit from Beeswax. However, these programs differ in two important ways that might necessitate extensions to Beeswax:

First, they often operate on shared data structure. For example, an eBPF program operating on file-system data is likely to need to traverse the directory structure, need to access or modify inodes and data blocks, etc. These data structures are shared with the kernel and for safety the programming model restricts how they are accessed. For example, XRP [72] programs use a runtime provided interface to translate logical blocks to physical blocks, and access data. While prefetching could also benefit such accesses, enabling it would require the runtime to expose multi-phase APIs for these shared structures. More generally, extending Beeswax to storage workloads may necessitate larger changes to the runtime and the interface between the kernel and eBPF programs. However, since eBPF interfaces for storage are still evolving these changes remain feasible.

Second, unlike network programs, eBPF programs in storage are often not triggered by I/O events. Instead, they are commonly invoked in response to system calls, as in XRP. This execution model limits opportunities to streamline batching as the runtime cannot naturally accumulate multiple inputs by waiting, as was the case with networking. Consequently, supporting Beeswax in this setting may require alternative batching or scheduling mechanisms to expose sufficient parallelism for overlapping computation with memory accesses.

Runtime support for batching. At runtime, the Beeswax library executes stages in a loop. Unfortunately, verifying loops is challenging, and therefore developers resort to unrolling loops or using helper functions (e.g., `bpf_loop`). Both are inefficient, and we found that loops are a significant source of overheads in Beeswax. Additional runtime support for batching could alleviate this problem: the runtime (which is written in native code and is not subject to the verifier's restrictions) can more efficiently loop between packets and stages. Of course, such a change would require baking Beeswax's multi-stage program design into the runtime, and thus might limit extensibility.

Impact on verifier. Our use of batching, multi-phase data structures, and multi-stage programs increase verification complexity. This is because these programs are larger (they include additional logic for data structure operations, going between stages, etc.), have more complex control flow graphs, and require more state (to track per-packet state and the current stage). This increases verification time alongside the chances that the program hits verifier complexity limits and is then rejected. In fact, we found that this was the case in some of our tests: we ran into cases where splitting a program into additional stages (without other changes) would cause the verifier to reject the program.

Fortunately, the verification algorithm used by the in-kernel eBPF verifier includes heuristics based on common patterns. The verifier can therefore be adopted to account for the Beeswax programming pattern alleviating this issue. We have not investigated the effort required for this, and will do so in the future.

9 Conclusion

People want to use I/O applications that have high performance and are easy to deploy. eBPF programs are popular because they are in the sweet spot for this: they have I/O overheads comparable to kernel-bypass libraries but without the associated fuss. In particular, they can be run on servers with other applications, with minimal reconfiguration. This is possible because of the restrictions imposed by eBPF's programming model.

However, the eBPF programming model also limits application performance. As we have shown in this paper, it is nearly impossible to use standard techniques to reduce the overhead of cache misses. Therefore, one might despair that we need to choose between performance optimization and ease of deployment. In this paper we have shown that this is not the case. Modest extensions to the eBPF runtime accompanied by a simple programming recipe allows eBPF programs to hide cache miss overheads. Furthermore, our extensions do not change the eBPF runtime's guarantees, and thus do not affect deployability.

More broadly, our results suggest that eBPF's performance envelope can be extended through targeted, incremental enhancements. Beyond this work, additional opportunities, such as finer-grained inlining control, and richer memory-access abstractions, could further narrow the gap between eBPF and specialized high-performance I/O frameworks.

Acknowledgments

We thank our anonymous shepherd and reviewers for their insightful comments. This work was partially supported by research grants from NEC Laboratories Europe, Google and the European Union  . The views and opinions expressed are those of the author(s) only and do not necessarily reflect those of the European Union or the European Health and Digital Executive Agency. Neither the European Union nor the granting authority can be held responsible for them. Thanks to Maria for the title.

References

[1] 2016. Mutilate: memcached load generator designed for high request rates. https://github.com/ix-project/mutilate.

[2] 2025. Swiss Tables Design Notes. https://abseil.io/about/design/swisstables.

[3] 2026. eBPF Documentation – Map type BPF_MAP_TYPE_PERCPU_ARRAY. https://docs.ebpf.io/linux/map-type/BPF_MAP_TYPE_PERCPU_ARRAY/.

[4] 2026. eBPF Documentation – What is eBPF?: Hook Overview. https://ebpf.io/what-is-ebpf/#hook-overview.

[5] Anastasia Ailamaki, Erietta Liarou, Pinar Tozun, Danica Porobic, and Iraklis Psaroudakis. 2017. Minimizing Memory Stalls. In *Databases on Modern Hardware: How to Stop Underutilization and Love Multicores*. Springer.

[6] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. (2012).

[7] Tom Barbette, Cyril Soldani, and Laurent Mathy. 2015. Fast userspace packet processing. In *Symposium on Architectures for Networking and Communications Systems (ANCS)*. ACM/IEEE.

[8] Ashish Bijlani and Umakishore Ramachandran. 2019. Extension framework for file systems in user space. In *Usenix Annual Technical Conference (ATC)*. USENIX.

[9] Laurin Brandner, Ayush Mishra, Sebastiano Miano, Aurojit Panda, Gianni Antichi, and Laurent Vanbever. 2026. Offloading L7 Policies to the Kernel. arXiv:2605.31084 [cs.NI]

[10] Matthew Butrovich. 2024. On Embedding Database Management System Logic in Operating Systems via Restricted Programming Environments. *Ph.D. Thesis, School of Computer Science, Carnegie Mellon University* (2024).

[11] Paul Chaignon. 2021. The Cost of BPF Tail Calls. https://pchaigno.github.io/ebpf/2021/03/22/cost-bpf-tail-calls.html.

[12] Sourav Chakraborty, N. V. Vinodchandran[1], and Kuldeep S. Meel. 2022. Distinct Elements in Streams: An Algorithm for the (Text) Book. *Leibniz International Proceedings in Informatics (LIPIcs), Volume: 244.*

[13] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. 2007. Improving hash join performance through prefetching. In *ACM Transaction on Database Systems, Volume: 32, Issue: 3.* ACM.

[14] Tien-Fu Chen and Jean-Loup Baer. 1992. Reducing memory latency via non-blocking and prefetching caches. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM.

[15] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. 1999. Cache-conscious structure layout. In *Conference on Programming Language Design and Implementation (PLDI)*. ACM.

[16] Cilium. 2026. Tetragon, eBPF-based Security Observability and Runtime Enforcement. https://tetragon.io.

[17] Cloud Native Computing Foundation. 2023. Cilium: Cloud native solution for networking, observability, and security. https://cilium.io/. Accessed: 2026-02-06.

[18] Cloudflare. 2020. Rakelimit, A fair-share ratelimiter implemented in BPF. https://github.com/cloudflare/rakelimit.

[19] Retina Contributors. 2026. Retina: kubernetes network observability platform. https://retina.sh/docs/Introduction/intro#introduction.

[20] Arnaldo Carvalho de Melo and Red Hat. 2010. The New Linux ' perf ' Tools. https://api.semanticscholar.org/CorpusID:10296207

[21] Abhishek Dhamija, Balasubramanian Madhavan, Hechao Li, Jie Meng, Shrikrishna Khare, Madhavi Rao, Lawrence Brakmo, Neil Spring, Prashanth Kannan, Srikanth Sundaresan, and Soudeh Ghorbani. 2024. A large-scale deployment of DCTCP. In *Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX.

[22] DPDK 2024. DPDK: Data Plane Development Kit. https://www.dpdk.org/.

[23] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Annual Technical Conference (ATC)*. USENIX.

[24] Kumar Kartikeya Dwivedi, Rishabh Iyer, and Sanidhya Kashyap. 2024. Fast, Flexible, and Practical Kernel Extensions. In *Symposium on Operating Systems Principles (SOSP)*. ACM.

[25] Arthur Fabre. 2018. L4Drop: XDP DDoS Mitigations. https://blog.cloudflare.com/l4drop-xdp-ebpf-based-ddos-mitigations/.

[26] Bolaji Gbadamosi, Luigi Leonardi, Tobias Pulls, Toke Høiland-Jørgensen, Simone Ferlin-Reiter, Simo Sorce, and Anna Brunström. 2024. The eBPF Runtime in the Linux Kernel. arXiv:2410.00026 [cs.OS]

[27] Hamid Ghasemirahni, Alireza Farshin, Mariano Scazzariello, Gerald Q. Maguire, Dejan Kostić, and Marco Chiesa. 2024. FAJITA: Stateful Packet Processing at 100 Million pps. In *International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*. ACM.

[28] 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 *Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX.

[29] Jörn-Thorben Hinz, Vamsi Addanki, Csaba Györgyi, Theo Jepsen, and Stefan Schmid. 2023. TCP's Third Eye: Leveraging eBPF for Telemetry-Powered Congestion Control. In *Workshop on EBPF and Kernel Extensions*. ACM.

[30] Intel Corporation. 2023. *Intel® 64 and IA-32 Architectures Optimization Reference Manual: Volume 1, §9.5.2.* Intel Corporation.

[31] Rishabh Iyer, Katerina Argyraki, and George Candea. 2024. Automatically Reasoning About How Systems Code Uses the CPU Cache. In *Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX.

[32] Jinghao Jia, Ruowen Qin, Milo Craun, Egor Lukiyanov, Ayush Bansal, Minh Phan, Michael V. Le, Hubertus Franke, Hani Jamjoom, Tianyin Xu, and Dan Williams. 2025. Rex: closing the language-verifier gap with safe and usable kernel extensions. In *Usenix Annual Technical Conference (ATC)*. USENIX Association, USA.

[33] Anuj Kalia, Dong Zhou, Michael Kaminsky, and David G. Andersen. 2015. Raising the Bar for Using GPUs in Software Packet Processing. In *Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX.

[34] Gyuyeong Kim and Dongsu Han. 2025. A Memory Pool Allocator for eBPF Applications. In *Workshop on EBPF and Kernel Extensions*. ACM.

[35] Don Knuth. 2023. The CVM Algorithm for Estimating Distinct Elements in Streams. https://cs.stanford.edu/~knuth/papers/cvm-note.pdf.

[36] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. 2012. When Prefetching Works, When It Doesn't, and Why. In *Transaction on Architecture and Code Optimization, Volume: 9, Issue: 1.* ACM.

[37] Tamás Lévai, Balázs Edvárd Kreith, and Gábor Rétvári. 2023. Supercharge WebRTC: Accelerate TURN Services with eBPF/XDP. In *Workshop on EBPF and Kernel Extensions*. ACM.

[38] Tamás Lévai, Felicián Németh, Barath Raghavan, and Gábor Rétvári. 2020. Batchy: batch-scheduling data flow graphs with service-level objectives. In *Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX.

[39] Shuo Li, Steven Chien, Tianyi Gao, and Michio Honda. 2026. Remote TCP Connection Offload and Applications. In *Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX.

[40] Chang Liu, Byungchul Tak, and Long Wang. 2024. Understanding Performance of eBPF Maps. In *Workshop on EBPF and Kernel Extensions*. ACM.

[41] Zhihong Luo, Sam Son, Sylvia Ratnasamy, and Scott Shenker. 2024. Harvesting Memory-bound CPU Stall Cycles in Software with MSH. In *Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX.

[42] Mingsong Lv, Nan Guan, Jan Reineke, Reinhard Wilhelm, and Wang Yi. 2016. A survey on static cache analysis for real-time systems. In *Leibniz Transactions on Embedded Systems, Volume: 3, Issue: 1.* Dagstuhl.

[43] Jinsong Mao, Hailun Ding, Juan Zhai, and Shiqing Ma. 2024. Merlin: Multi-tier Optimization of eBPF Code for Performance and Compactness. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM.

[44] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Mauricio Vásquez Bernal, Yunsong Lu, and Jianwen Pi. 2019. Securing Linux with a faster and scalable iptables. In *SIGCOMM Computer Communication Review (CCR), Volume: 49, Issue: 3.* ACM.

[45] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Massimo Tumolo, and Mauricio Vásquez Bernal. 2018. Creating Complex Network Services with eBPF: Experience and Lessons Learned. In *International Conference on High Performance Switching and Routing (HPSR)*. IEEE.

[46] Sebastiano Miano, Xiaoqi Chen, Ran Ben Basat, and Gianni Antichi. 2023. Fast In-kernel Traffic Sketching in eBPF. In *SIGCOMM Computer Communication Review (CCR), Volume: 53, Issue: 1.* ACM.

[47] Sebastiano Miano, Alireza Sanaee, Fulvio Risso, Gábor Rétvári, and Gianni Antichi. 2024. Morpheus: A Run Time Compiler and Optimizer for Software Data Planes. In *Transactions on Networking, Volume: 3, Issue: 3.* IEEE/ACM.

[48] László Molnár, Gergely Pongrácz, Gábor Enyedi, Zoltán Lajos Kis, Levente Csikor, Ferenc Juhász, Attila Kőrösi, and Gábor Rétvári. 2016. Dataplane Specialization for High-performance OpenFlow Software Switching. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[49] Máté Nagy, Tamás Lévai, Felicián Németh, Aurojit Panda, Gianni Antichi, and Gábor Rétvári. 2026. Elastic Scaling of Real-Time Communication Services. In *IEEE Transactions on Network and Service Management, Volume: 23.* IEEE.

[50] Usama Naseer, Luca Niccolini, Udip Pant, Alan Frindell, Ranjeeth Dasineni, and Theophilus A. Benson. 2020. Zero Downtime Release: Disruption-free Load Balancing of a Multi-Billion User Website. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[51] Network Startup Resource Center. 2025. RouteViews. doi:10.7264/1Y7V-2D90

[52] Vladimiro Paschali, Andrea Monterubbiano, Francesco Fazzari, Sebastiano Miano, and Salvatore Pontarelli. 2025. X2DP: eXtended eXpress Data Path. In *Proceedings of the ACM SIGCOMM 2025 Posters and Demos*. ACM.

[53] Vladimiro Paschali, Andrea Monterubbiano, Francesco Fazzari, Michael Swift, and Salvatore Pontarelli. 2025. InXpect: Lightweight XDP Profiling. In *Workshop on EBPF and Kernel Extensions*. ACM.

[54] Hugo Sadok, Nirav Atre, Zhipeng Zhao, Daniel S. Berger, James C. Hoe, Aurojit Panda, Justine Sherry, and Ren Wang. 2023. Enso: A Streaming Interface for NIC-Application Communication. In *Symposium on Operating Systems Design*

and Implementation (OSDI). USENIX.
[55] Nicolas Le Scouarnec. 2018. Cuckoo++ hash tables: high-performance hash tables for networking applications. In *Symposium on Architectures for Networking and Communications Systems (ANCS)*. ACM.
[56] Raimund Seidel and Cecilia R. Aragon. 1996. Randomized search trees. In *Algorithmica, Volume: 16, Issue: 4*. Springer.
[57] Farbod Shahinfar, Sebastiano Miano, Aurojit Panda, and Gianni Antichi. 2025. Demystifying Performance of eBPF Network Applications. In *International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*. ACM.
[58] Farbod Shahinfar, Sebastiano Miano, Alireza Sanaee, Giuseppe Siracusano, Roberto Bifulco, and Gianni Antichi. 2021. The case for network functions decomposition. In *International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*. ACM.
[59] Nikita Shirokov and Ranjeeth Dasineni. 2018. Katran, a scalable network load balancer. https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer/.
[60] Franco Solleza, Justus Adam, Akshay Narayan, Malte Schwarzkopf, Andrew Crotty, and Nesime Tatbul. 2025. Kernel Extension DSLs Should Be Verifier-Safe!. In *Workshop on EBPF and Kernel Extensions*. ACM.
[61] Joel Sommers, Nolan Rudolph, and Ramakrishnan Durairajan. 2023. Schooling NOOBs with eBPF. In *Workshop on EBPF and Kernel Extensions*. ACM.
[62] SPDK 2025. SPDK: Storage Performance Development Kit. https://spdk.io/.
[63] Alexei Starovoitov. 2020. bpf: Introduce global functions. https://lore.kernel.org/bpf/20200109063745.3154913-6-ast@kernel.org/t/.
[64] Alexei Starovoitov. 2024. BPF Arena. https://lwn.net/Articles/961594/.
[65] William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. 2021. Revisiting the open vSwitch dataplane ten years later. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.
[66] Nikita Tyunyayev, Clément Delzotti, Haggai Eran, and Tom Barbette. 2025. ASNI: Redefining the Interface Between SmartNICs and Applications. *Proceedings of the ACM on Networking, Volume 3, Number: 2*.
[67] David Wragg. 2020. Unimog - Cloudflare's edge load balancer. https://blog.cloudflare.com/unimog-cloudflares-edge-load-balancer/.
[68] Qiongwen Xu, Michael D. Wong, Tanvi Wagle, Srinivas Narayana, and Anirudh Sivaraman. 2021. Synthesizing safe and efficient kernel extensions for packet processing. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.
[69] Bin Yang, Dian Shen, Junxue Zhang, Hanlin Yang, Lunqi Zhao, Beilun Wang, Guyue Liu, and Kai Chen. 2025. eNetSTL: Towards an In-kernel Library for High-Performance eBPF-based Network Functions. In *European Conference on Computer Systems (EuroSys)*. ACM.
[70] Rui Yang and Marios Kogias. 2023. HEELS: A Host-Enabled eBPF-Based Load Balancing Scheme. In *Workshop on EBPF and Kernel Extensions*. ACM.
[71] Tom Yates. 2017. Using eBPF and XDP in Suricata. https://lwn.net/Articles/737771/.
[72] 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 *Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX.
[73] Yang Zhou, Zezhou Wang, Sowmya Dharanipragada, and Minlan Yu. 2023. Electrode: Accelerating Distributed Protocols with eBPF. In *Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX.
[74] Yang Zhou, Xingyu Xiang, Matthew Kiley, Sowmya Dharanipragada, and Minlan Yu. 2024. DINT: Fast In-Kernel Distributed Transactions with eBPF. In *Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX.