An Expressive Query Interface for Kernel Telemetry

Richard Tang

Advisor: Malte Schwarzkopf
Reader: Ugur Cetintemel

Department of Computer Science
Brown University
Providence, RI
May 12, 2024
Contents

Acknowledgements .................................................. 4

Abstract ........................................................................ 5

1 Introduction ............................................................... 6
  1.1 BeeHouse ............................................................. 7

2 Background and Related Work ..................................... 8
  2.1 Observability ......................................................... 8
    2.1.1 Kernel Telemetry .............................................. 8
  2.2 eBPF ................................................................. 9
    2.2.1 eBPF Program Development .............................. 9
    2.2.2 Streaming Data Management ............................ 10
  2.3 eBPF Development Challenges ................................. 12
  2.4 Related Work ....................................................... 15
    2.4.1 bcc .......................................................... 15
    2.4.2 Cilium ........................................................ 15
    2.4.3 bpftrace .................................................... 16

3 BeeHouse Design ...................................................... 18
  3.1 BeeHouse API ..................................................... 18
  3.2 BPF Query Language (BQL) ................................. 19
  3.3 Data Representation .............................................. 21
    3.3.1 Relational Model .......................................... 21
    3.3.2 Query Output ............................................. 23
  3.4 Query Plans ...................................................... 23
    3.4.1 Physical Plan and Query Optimization ................. 23
  3.5 Code Generation .................................................. 25
  3.6 Query Execution and Post-Processing ....................... 25

4 Implementation ....................................................... 26
  4.1 BeeHouse Query Parsing ...................................... 26
  4.2 Code Generation ................................................ 26
  4.3 Prototype Limitations ......................................... 26

5 Evaluation .............................................................. 29
  5.1 End-to-End Evaluation ......................................... 29
  5.2 Evaluation of Computation Pushdown into the Kernel .... 31
  5.3 Performance Drilldown ........................................... 33
  5.4 Discussion ......................................................... 34
Acknowledgements

Thank you to my thesis advisor, Malte Schwarzkopf, for supporting me throughout not just my work on this thesis, but throughout all four years of undergrad. From my first class—CS300—with Malte, to the subsequent years (H)TAing and researching under him, I have learned so much and developed as a person thanks to him. Thank you for the continued support and mentorship throughout my time in college.

Thank you to Franco Solleza for onboarding me to Mach, then to this new eBPF query language project. I’ve had the amazing opportunity to work on this project with Franco and learn about databases, systems, and research from him. Franco has offered invaluable support, guidance, and expertise on every step of this thesis, and I am so grateful for his mentorship.

Thank you for all of the professors, friends, and family who have supported me throughout my time at Brown. You have made these past four years unforgettable, and I am so grateful for everyone—current and graduated—who has positively impacted my life: Emily, Jessica, Kiara, Michael, Orion, my roommates, and so many more.

Thank you.
Abstract

To understand the complex interactions in modern software, engineers often rely on detailed kernel telemetry. Increasingly, developers have turned to eBPF (the extended Berkeley Packet Filter) for kernel data collection, as it provides an extensible framework for high performance, low overhead instrumentation. However, writing eBPF programs is notoriously difficult for developers, and existing programs are limited and frequently inefficient.

We introduce BeeHouse, an eBPF query engine that enables performant kernel data collection via an expressive, high-level interface. BeeHouse provides a familiar relational data model over existing kernel tracing infrastructure, allowing developers to query arbitrary kernel events with low overhead. Internally, BeeHouse processes SQL queries by generating and optimizing an eBPF program to execute the query, then streaming the output via a structured schema definition.

We evaluate BeeHouse-generated eBPF programs on a RocksDB case study simulating a real-world workload. BeeHouse queries incur a low abstraction overhead versus hand-optimized queries (3.2% vs. 1.7%), and offer a $5 - 6 \times$ performance improvement over a baseline eBPF program similar to what a non-expert engineer would write.
1 Introduction

As modern software continues to grow in complexity, observability and continuous monitoring become increasingly essential in ensuring a system’s health and performance. Telemetry data like metrics, logs, and traces provide valuable insights: application-level logs on error messages and access patterns can aid in debugging software bugs and identifying performance regressions; metrics on resource usage (e.g. CPU, memory, and I/O), tail latencies, and uptime track overall health and reveal high-level anomalies in the system; and distributed traces follow execution flow and identify bottlenecks within a request processing pipeline.

Combined, these types of telemetry data provide application-level monitoring of system health. However, while this telemetry data can reveal high-level symptoms of system anomalies, there are often insufficient to pinpoint the root cause, as they lack the requisite granularity and thus do not contain crucial information needed for debugging.

Concretely, consider a performance engineer investigating a performance regression in a microservices application. Via application-level aggregated metrics, the engineer can identify a system bottleneck from the backing RocksDB application by identifying spiking tail latency metrics; from there, they can use traces or additional instrumentation to pinpoint the specific problematic function (say, pread64 within the GET operation). However, although the engineer now knows where the system bottleneck arises, they do not know why it is occurring.

To analyze the root cause of performance regressions or system anomalies, developers must turn to kernel data, collected from kernel events with a much higher level of granularity from the event. Using kernel data, developers can interactively investigate and identify the root cause of system anomalies. In the above example, an engineer can use kernel data to formulate a hypothesis about the root cause (for instance, they could correlate system call latency with other kernel events like page cache events, and identify that a competing process is causing repeated page cache evictions).

However, actually generating kernel data can be highly involved, and the existing kernel functionality can be inflexible, inefficient, and difficult to use: tracing infrastructure often relies on costly interrupt-based event instrumentation, requires extensive kernel knowledge in order to develop efficient and sound programs, and can necessitate kernel patches [64, 57], greatly complicating development and maintenance for developers.

The growth and development of eBPF [26], a kernel subsystem, enables an extensible interface for dynamic kernel tracing. eBPF provides a sandboxed virtual environment to run statically verified custom user “probes” that can perform in-kernel processing and context-specific information retrieval. These probes are then run at user-specified events, from kernel tracepoints and kprobes to the network ingress/egress path.

eBPF facilitates flexible kernel data collection; unfortunately, program development can be prohibitively complex. Developers must grapple with not only kernel structures and contexts, but also subtleties in the eBPF architecture—in particular, the eBPF program
verifier—that make writing eBPF programs difficult for application developers. Without a thorough knowledge of eBPF, developers often resort to simple but inefficient programs, or existing but limited tools (e.g. from bcc/bpftrace [8, 52]).

1.1 BeeHouse

To ease kernel data collection, we propose BeeHouse, an eBPF query engine. BeeHouse allows users to declaratively query for kernel telemetry, without needing to develop any custom tracing programs. At a high level, BeeHouse takes an input query from the user, parses and generates a query physical plan consisting of a kernel-space (eBPF) event processing component and a userspace post-processing component, then emits record batches back to the user.

BeeHouse has three design goals:

1. **Provide an expressive query interface** for application developers to dynamically query for kernel data at a high level, abstracting away internal eBPF implementation details such that a deep knowledge of eBPF or the kernel is not required.

2. **Expose a general, structured API** for generated kernel data to enable seamless integration with streaming data analytics pipelines like Spark [67], or existing observability systems like Mach [59] or M3DB [65].

3. **Facilitate performance optimizations** by exploring the impacts of user-kernel space transitions in physical plans and evaluating the impact of traditional RDBMS optimizations on BeeHouse query plans.

We implement a BeeHouse prototype in Rust, and evaluate its performance on a simulated RocksDB [53] workload. We find that BeeHouse’s abstraction layer incurs a low—and resolvable—overhead over hand-optimized eBPF programs (3.2% vs 1.7%), and outperforms existing methods of eBPF-based kernel data collection by 5 – 6×.

In summary, this thesis makes the following contributions:

1. We propose a **streaming data model** over existing kernel event streams by associating events with a structured relation and extending SQL to support streaming semantics.

2. We **dynamically generate and load eBPF tracing programs** at runtime in a composable way, lifting the overhead of developing custom eBPF programs on developers.

3. We **investigate the impact of computation pushdown into the kernel** on runtime overhead in eBPF contexts.
2 Background and Related Work

2.1 Observability

In order to develop and maintain robust systems, developers increasingly rely on observability systems to monitor system health and performance [62, 10].

Current observability systems focus primarily on three types of telemetry data: logs, metrics, and traces (collectively termed the “Three Pillars of Observability” [62, ch. 4]):

- **Logs** are semi-structured or unstructured strings added by developers to expose information with local context, such as stack traces or access events.

- **Metrics** provide quantitative measurements at a specific time as either counters of cumulative values (e.g. GC collection executions), gauges of system state (e.g. CPU usage), or histograms of observed values (e.g. request latencies) [44].

- **Traces** follow program execution flow and resemble call graphs.

Current observability systems are primarily geared towards these three types of telemetry; however, they often fail to capture the underlying root cause of system anomalies, as they rely almost entirely on application-level instrumentation [54] and capture primarily aggregate metrics. As a consequence, they often miss more specific information critical for problem diagnosis.

2.1.1 Kernel Telemetry

For finer-grained data, developers often turn to the kernel for specific event data, such as page cache evictions or CPU scheduler events. Since the data originates from the kernel itself, the data contains comprehensive information about the context from which it originated (e.g. inode and device number for page cache eviction).

Because kernel data collection primarily serves to debug system anomalies and performance regressions, the collection process faces strict requirements. Especially since this functionality is often injected at program hot paths or in systems under high load, kernel data collection programs must incur negligible overhead and remain performant, even under intense resource pressure. In addition, kernel data must be as complete as possible, as sparse data sampling can result in key anomalous events frequently being dropped.

To support kernel data collection, the Linux kernel exposes `tracefs` [1], a thin layer over kernel events; moreover, tools from the research community [57, 39, 66] provide varying amounts of application tracing utilities.

- Event profilers like `perf` [50], `ftrace` [28], and `SystemTap` [64] trace kernel events; however, these systems incur prohibitive overhead costs that make them unsuitable for production systems [11, 39].
• Tracepoints [24] and Kprobes [23] from tracefs allow developers to instrument specific kernel instructions or events with low overhead. However, tracepoint interactions occur solely in user-space, requiring each kernel context to be emitted to user-space and incurring significant overhead as the number of events increase, and kprobes introduce significant safety risks as faulty programs could crash the kernel [43].

• Tracing instrumentation tools like Magpie [7], KUTrace [57], Shim [66], and Hubble [39] collect kernel data through lightweight instrumentation. However, they are purpose-specific: for instance, Hubble specifically targets the Android runtime, and Shim primarily targets hardware performance counters and signals.

Moreover, each independent system contains adhoc formats, and must be combined with other tools for a complete picture. This tool fragmentation leads to a manual, unscalable process of actively monitoring and synthesizing several data sources, and poses real difficulties for developers trying to debug anomalies. In a case study by Cloudflare [40, 41] debugging a latency spike, five separate custom SystemTap scripts were required, alongside netstat and tcpdump, in order to identify the root cause.

2.2 eBPF

A recent kernel technology, the extended Berkeley Packet Filter ([e]BPF), provides a promising approach to kernel data collection by providing a minimalistic sandboxed “virtual machine” [21, 12] to execute custom user programs, allowing developers to safely and efficiently extend kernel capabilities without kernel patches or modules. eBPF’s integration with tracefs facilitates custom kernel logic at tracepoints and kprobes, making it a flexible and expressive tool for kernel data collection. Beyond tracing, eBPF has also been used for efficient in-kernel packet processing [34], CPU scheduling [35], and kernel bypass storage functions [68].

2.2.1 eBPF Program Development

To write an eBPF program, developers write code in essentially a subset of C, with additional restrictions (for example, the stack size is restricted to 512 bytes, a maximum of 1 million instructions are allowed, and dynamic memory is forbidden). LLVM/clang then compiles the program into BPF bytecode. The eBPF verifier ensures soundness of eBPF programs, rejecting potentially unsound programs by checking for guaranteed termination and memory safety. The verifier traverses through all program paths, using heuristics to prune potential program branches [30] and maintaining a DAG to ensure bounded loop termination and other CFG validation [27].

After verification, BPF loads the bytecode into the designated hook point. Then, every kernel event triggers the BPF program within the context; each event exposes specific context, plus generic kernel attributes (e.g. pid, ktime, etc). After processing the context,
BPF program can then communicate to userspace by writing data to a shared BPF map. Although recent developments [47] optimize the data transfer, it still incurs significant overhead: BPF must synchronize ringbuf access, the kernel must interrupt the polling user-space process and switch contexts, and the user-space process must copy the data from the BPF map.

As a concrete example, Figure 1 contains a standard BPF program to instrument pread64 system call invocations by recording the ktime, fd, and CPU id on which the call was invoked. At each annotated location in the code:

1. Defines a struct representation of transferred data to simplify access over raw bytes.
2. Defines a ringbuf BPF map to communicate data between user and kernel space.
3. Defines the actual BPF program; the SEC macro denotes the kernel event to instrument.
4. Allocates space in the ring buffer to be communicated to user-space.
5. Checks for pointer validity; the verifier needs such bounds and null checks to validate program soundness.
6. Fetches struct values from both the specific event context and the generic kernel context.
7. Submits the data to the ringbuf for the userspace process to read.

### 2.2.2 Streaming Data Management

eBPF’s event-driven architecture causes kernel events to closely resemble data streams: because programs trigger only when a kernel event occurs, the resulting output models a data stream with an unbounded, continuous collection of data from the kernel. Thus, we briefly explore basic concepts in streaming data management.

A data stream is a real-time, continuous, ordered (here, by timestamp) sequence of items [29]. Queries over data thus run continuously over a period of time, and incrementally return new results as new data arrive [18, 38].

Data stream management and processing introduces novel considerations:

- A standard relational data model cannot be directly applied to continuous queries over data streams.
- Complete streams cannot be stored, requiring stateful synopses of a portion of the stream and/or approximate summary sketch data structures.
- Streaming query plans cannot directly use blocking operators that must consume the entire input before results are produced.
Figure 1: A standard pread64 tracing eBPF program to record the syscall invocation’s ktime, fd, and cpu.
To manage the continuous, unbounded aspect of streams, a fundamental stream operator is the *window*, which discretizes the stream into the latest partial view of the stream. Windows can be either count-based, storing up to \( N \) elements, or time-based, storing the past \( N \) units of time (e.g. 10ms). The window moves forward based on some parameter \( \text{step} \); if \( N = \text{step} \), the window is a **tumbling** window. In a query context, the window operator converts the unbounded stream into a bounded relation, allowing standard stateful operations like aggregations, joins, etc. over the window. To avoid space requirements linear in the window size, stateful operations sometimes employ *sketches*, or probabilistic approximation data structures, that provide rigorous accuracy guarantees. Sketches support a variety of measurement tasks, such as heavy hitters detection [9, 19, 58], frequency estimation [17, 22], and counting distinct elements [5, 6].

Within the context of eBPF, we focus primarily on traditional stream processing techniques. In particular, eBPF’s restricted feature set—specifically, its hard memory and instruction limits, and lack of dynamic memory—make certain streaming operations difficult or impossible to implement (e.g. arbitrary joins), requiring careful investigation of what work can be delegated to kernel-space eBPF programs, and what work must be implemented in user space.

### 2.3 eBPF Development Challenges

Despite its various benefits, onboarding into the eBPF ecosystem and developing performant, verifiable eBPF programs remain a significant challenge.

First, the eBPF verifier can reject seemingly sound eBPF programs. Due to its heuristic-based verification scheme, the verifier can be overly restrictive, rejecting programs due to an inability to completely verify the program logic; in other cases, the verifier encounters constructs outside of eBPF’s current supported feature set, causing it to reject programs. To complicate matters, the verifier’s error messages deal primarily with eBPF bytecode and the virtual registers, and can be uninformative at best and misleading at worst. We consider three illustrative examples:

1. The eBPF program in Figure 2 accumulates up to RINGBUF_MAX_ENTRIES context values, then emits them to the ring buffer.

   ```c
   // ... more output ... //
   R1_w = inv262144 R2_w = inv(id = 0, umin = 262144) R10 = fp0
   ; u64 *records = bpf_ringbuf_reserve(&rb, count * sizeof(u64), 0);
   7: (67) r2 <<= 3
   8: (b7) r6 = 0
   12: (85) call bpf_ringbuf_reserve #131
   R2 is not a known constant
   ```
Figure 2: An example program that accumulates values before emitting to user-space.

From a developer standpoint, we can verify that the eBPF program is safe to execute, as `global_count` will only ever be at most `RB_MAX_ENTRIES`, and because we assign `global_count` to a local variable `count`, this would not pose an issue with other concurrently executing processes. However, because the verifier operates on variable ranges with only individual execution-level context, it is unable to verify the logic.

To remedy this, a line manually setting `count = RB_MAX_ENTRIES` would be required. This kind of massaging to appease the verifier is common in eBPF programs.

2. The eBPF code snippet in Figure 3 attempts to copy over data from one buffer to another (both stored as global variables). On program load, the eBPF verifier emits the message, `error: A call to built-in function 'memcpy' is not supported`, and rejects the program, even though the clang intrinsic `__builtin_memcpy`
is supported in eBPF environments. After much digging, the cause of this error is because the stack size is limited to 512B, and so builtin memory operations fail when operating on structs larger than that [36].

To remedy this, the `memcpy` would have to be replaced with a call to `bpf_probe_read_kernel`, which introduces an additional overhead of memory copying and runtime safety checks.

3. The eBPF code snippet in Figure 4 attempts to declare a map of $2^{20}$ elements.

```c
struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __type(key, u64);
    __type(value, u64);
    __uint(max_entries, (1 << 20));
} map SEC(".maps");
```

Figure 4: An eBPF code snippet attempting to initialize a large map.

On program load, the BPF verifier emits the error message:

```
Error in bpf_create_map_xattr(prog.bss):Argument list too long (-7). Retrying without BTF.
map 'prog.bss': failed to create: Argument list too long(-7)
libbpf: failed to load object 'prog_bpf'
libbpf: failed to load BPF skeleton 'prog_bpf': -7
```

After much investigation, the root cause for this is because `bpf_create_map` invokes the internal kernel `kmalloc`, which on most machines has a limit of 4MB; thus, maps larger than that are rejected. Unfortunately, the actual error message provides little insight into BPF’s limitations, with no specific indication that map sizes have this fixed limit.

There are some important takeaways from this. First, due to the verifier’s limited individual-execution scope, developers must frequently appease the verifier by adding redundant checks and redundant accesses, incurring real performance hits; otherwise, sound programs would be rejected. Second, the verifier often emits obscure or misleading error messages, forcing the developer to manually test different parts of their program, scour online discussion forums, or investigate the BPF kernel itself to pinpoint the cause. These combined can significantly impede developer workflows.

Beyond the verifier, the eBPF ecosystem also can hinder developers. Due to its volatile and evolving interface, APIs experience frequent (and often breaking) changes; moreover, online documentation often fails to keep up with recent developments, resulting in misleading and outdated information, or even a complete lack thereof, forcing developers to resort to kernel patch updates or mailing lists. As concrete examples, the `bpf-helpers` manpage explicitly refers developers to the kernel source for an up-to-date list of helper functions [15];
and simple questions such as “Can I clear a BPF map within an eBPF program?”, “How can I pin BPF maps to share across programs?”, and “Can I use this BPF helper function at this kernel event?” require deep exploration of online resources or even the kernel source [33, 63].

Moreover, due to the eBPF architecture, developers can easily create inefficient programs without a deep knowledge of eBPF subtleties. eBPF structures often incur hidden synchronization overhead due to its concurrent execution environment; different per-CPU and task-local constructs can reduce overhead at the expense of higher developer complexity, but only if developers are intimately aware with the environment [25].

2.4 Related Work

To ease eBPF development, a robust ecosystem of tools aim to provide higher-level wrappers around eBPF internals.

2.4.1 bcc

One of the most popular development frameworks is the BPF Compiler Collection (bcc) [8]. Development follows a similar workflow to raw BPF program development, with some key additions:

- bcc introduces a higher-level Python/Lua frontend for communicating with BPF programs and the bpf syscall, simplifying access and modification of shared BPF maps, global variables, and other program configuration options.

- bcc adds additional macros and pre-processing steps before loading BPF code, which simplifies section/program type macros, map definitions and accesses, and certain BPF helper functions.

Despite its simplified development environment, however, bcc runs into numerous challenges. First, although some constructs are simplified, the actual BPF program—the key instrumentation code—remains in BPF C, incurring the same developer overhead. Additionally, bcc’s various abstractions can further hinder development, as it employs idiosyncratic naming conventions, hides various initialization and auto-generated struct definitions, and uses a non-standard object-oriented version of C that often differs from internal kernel operations [46]. As a result, the developer community often recommends using the standard libbpf development environment.

2.4.2 Cilium

Cilium is a cloud-native monitoring, networking, and security platform for managing containerized Kubernetes environments [20]. Cilium provides useful utilities out-of-the-box, exposing Prometheus metrics for container and pod health, and abstracts away much of the complexity of eBPF by using it only internally.
However, it focuses on cloud, containerized environments, with functionality geared primarily towards networking and security; as such, its monitoring utilities mainly target those purposes, and due to its higher-level nature, exposes more aggregate metrics on system health, rather than the high-granularity kernel data needed for root cause analysis.

In addition to the Cilium platform, it provides a Go frontend wrapping eBPF functionality (much like bcc), simplifying management of BPF maps and programs from user-space. However, like bcc, Cilium also relies on embedded eBPF programs. Thus, the burden of developing BPF instrumentation programs still lies with the developer.

2.4.3 bpftrace

bpftrace [52] provides the most high-level interface over eBPF, exposing a tracing language similar to DTrace [16]. bpftrace takes in a script written in its higher-level language, parses it into an AST, converts it into LLVM IR, then hooks into the LLVM toolchain to compile into eBPF bytecode. bpftrace then automatically loads the compiled program into the kernel.

bpftrace, in many ways, provides a sufficiently high-level abstraction over eBPF, enabling developers to write custom tracing scripts without writing BPF C. However, bpftrace is a fundamentally procedural language; thus, developers must explicitly enumerate program steps and manually manage BPF state, grounding development into the same procedural programming required for BPF C. For instance, in the bpftrace program in Figure 5, a developer must operate on the arguments in a C-like manner (1), and manually manage BPF map state (2).
tracepoint:syscalls:sys_enter_open,
tracepoint:syscalls:sys_enter_openat {
    // 2
    @filename[tid] = args.filename;
}

tracepoint:syscalls:sys_exit_open,
tracepoint:syscalls:sys_exit_openat /
@filename[tid]/ {
    // 1
    $ret = args.ret;
    $fd = $ret >= 0 ? $ret : -1;
    $errno = $ret >= 0 ? 0 : -$ret;
    printf("%-6d %-16s %4d %3d %s\n", pid, comm, $fd, $errno,
        str(@filename[tid]));
    delete(@filename[tid]);
}

END {
    // 2
    clear(@filename);
}

Figure 5: The bpftrace opensnoop.bt script [49]
3 BeeHouse Design

We now introduce BeeHouse, an eBPF query engine that facilitates accessible kernel data collection. Motivated by the challenges in kernel data collection and eBPF program development described in chapter 2, BeeHouse has three high-level design goals:

1. **Provide an expressive, accessible query interface** for developers to dynamically collect kernel data on running applications through a familiar relational query interface. The underlying eBPF infrastructure should be abstracted away from developers as much as possible, lowering the entry pre-requisites for rich kernel data collection.

2. **Expose a general, structured output API** for generated kernel data to enable seamless integration with existing data analytics pipelines or observability systems. Developers should be able to easily hook BeeHouse query results into a separate platform for post-processing or storage.

3. **Facilitate performance optimizations** by exploring the impacts of user-kernel space transitions in physical plans and evaluating the impact of traditional RDBMS optimizations on BeeHouse query plans.

![Figure 6: The BeeHouse system architecture. BeeHouse transforms a BQL query (§3.2) into an abstract syntax tree, analyzes the query, then generates an eBPF program to execute the query. The program is then loaded into the kernel, and post-processed kernel data is streamed to the user.](image-url)

BeeHouse transforms high-level BQL queries through a series of steps before generating an eBPF tracing program, loading it into the kernel, and streaming output results to the user (Figure 6). Crucially, BeeHouse abstracts away the internals of eBPF from the user, enabling kernel data collection in a declarative manner.

3.1 BeeHouse API

From a client perspective, BeeHouse exposes a query executor API with two methods:

The executor takes in a BQL query (§3.2) and converts it into an eBPF instrumentation program executing in the kernel. If successful, the executor returns the query’s schema
### Function Description

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>execute_query(bql_query: String) -&gt; (Schema, Stream&lt;RecordBatch&gt;)</code></td>
<td>Executes a BQL query, streaming results back to the user in a defined schema.</td>
</tr>
<tr>
<td><code>get_query_stats(query: String) -&gt; QueryStats</code></td>
<td>Fetches the stats of a specific query.</td>
</tr>
</tbody>
</table>

Table 1: BeeHouse API.

definition (§3.3.2), and a receiver channel of streamed record batches (i.e., kernel data after being processed by the query).

### 3.2 BPF Query Language (BQL)

To enable declarative querying, BeeHouse exposes BQL, a query language lightly extended from SQL. At its core, BQL supports the standard SQL relational algebra, and uses its logical operator set to express typical data manipulation. However, BQL extends standard SQL to support streaming semantics using a Window operator, and adds additional support for time-series analysis via histograms and quantiles.

BQL’s design takes heavily from the Continuous Query Language (CQL) from Stanford’s Data Stream Management System [2, 3]. Table 2 provides a complete logical operator list in BQL.

A time-ordered stream (in this context, the ktime == clock_gettime(CLOCK_MONOTONIC)) represents each kernel event. The Window operator converts the kernel event stream into a bounded relation by providing a tumbling window over the stream (that is, the current window represents a discrete relation). BQL operators operate on streams and/or relations, depending on their type:

- Stateless operators like project, filter, and map operate on either streams or relations, as each operates only on individual events and thus do not require some bounded amount of events to compute.

- Ungrouped stateful aggregations operate on streams, accumulating values until the query is finished. This is permitted due to the minimal amount of state needed to store the supported aggregation values: for instance, a count aggregation without grouping requires only a constant amount of space—one u64—regardless of the amount of elements in the stream.

- Stateful (grouped) aggregations also operate on bounded (i.e. windowed) relations. Data manipulation with these constructs performs the same functionality as their standard SQL equivalents: for instance, the max computed over a 1024-count window is functionally equivalent to a max computed over a 1024-row relation.

- Joins currently only operate on relations: they must compute the result on the entire input before results are produced, so the input sources must be bounded relations, not unbounded streams. BQL does not currently support streaming joins, as they require
Table 2: Logical operators used in BQL.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select</td>
<td>Selects a stream from a kernel tracing event, such as <code>tracepoint/filemap/mm_filemap_add_to_page_cache</code>.</td>
</tr>
<tr>
<td>Window</td>
<td>Partitions unbounded streams into bounded relations, either by time or tuple count. Currently, BQL only supports tumbling windows (i.e. those with the same size and step), since the verifier cannot handle the large loops that might occur in stepping windows (e.g. for some operators, stepping forward $N$ events requires $N$ loop iterations; as $N$ grows large, the verifier takes prohibitively long before failing).</td>
</tr>
<tr>
<td>Project</td>
<td>Projects specific attributes from a stream/relation. These attributes may be event-specific (e.g. <code>pfn</code> for page cache evictions) or generic system attributes (e.g. <code>pid/tgid</code>, <code>cgroup</code>, or <code>CPU/SMP id</code>).</td>
</tr>
<tr>
<td>Filter</td>
<td>Filters attributes based on some conditional statement, such as <code>pid == 12000</code>, or <code>count &gt;= 4096</code>.</td>
</tr>
<tr>
<td>Map</td>
<td>Maps attributes from one value to another. Example maps use basic arithmetic (e.g. <code>count * 2</code>) or a pre-defined set of BPF helper functions (e.g. <code>bpf_get_ns_current_pid_tgid(dev, ino)</code>).</td>
</tr>
<tr>
<td>GroupBy</td>
<td>Groups events according to a set of grouping keys (e.g. <code>(fd, cpu)</code>).</td>
</tr>
<tr>
<td>Aggregate</td>
<td>Performs some aggregation over grouped elements. eBPF and prototype limitations currently restrict the supported aggregations to <code>max</code>, <code>min</code>, <code>sum</code>, <code>average</code>, <code>count</code>, <code>histograms</code>, and <code>quantiles</code>.</td>
</tr>
<tr>
<td>Distinct</td>
<td>Eliminates duplicates according to the group by key, preserving the most recent event.</td>
</tr>
<tr>
<td>Join</td>
<td>Joins two event streams by some specified condition (e.g. <code>A.pid == B.pid</code>).</td>
</tr>
</tbody>
</table>
external state management to determine join logic and previously seen records (that in
the worst case, could be unbounded), a condition not currently supported by eBPF.

From a user standpoint, besides from the additional Window operator to accommodate
the streaming nature of kernel events, BQL’s query language models the same semantics as
standard SQL with only slight syntactical differences.

As an example, Figure 7 shows a continuous aggregation query over pread64 syscall
invocations.

```
SELECT fd, cpu, COUNT(*), MAX(count), AVG(count)
    FROM tracepoint/syscalls/sys_enter_pread64
    GROUP BY fd, cpu
    WHERE pid == 1041370
    WINDOW(time, 1000, 1000);
```

Figure 7: A continuous BQL aggregation query to aggregate information from the pread64
invocation tracepoint. The additional Window operator converts the event stream into a
relation: the first argument takes in the window type (time or count), the second argument
the interval, and the third argument the step size (for time, in milliseconds).

The query establishes a tumbling 1 second time window over the sys_enter_pread64
tracepoint, filters for syscalls from only the specified pid (here 1041370), then groups all
invocations on the same fd and from the same cpu, and computes three aggregations: the
total number of elements, the maximum number of bytes read, and the average amount of
bytes read (for context, count in the pread64 syscall represents the amount of bytes to
read).

The query then returns events every time the window tumbles (here, every second) as a
RecordBatch (Section 3.3.2).

3.3 Data Representation

3.3.1 Relational Model

Within BeeHouse, each kernel event is modeled as a streaming relation with a fixed set of
attributes.

The existing kernel infrastructure inherently exposes a relational structure for its tracing
events: all (raw) tracepoints follow a fixed format (in Linux, defined at /sys/kernel/tracing
/events/<category>/<name>/<format>), while u[ret]probes and k[ret]probes expose
some probe-specific context with a general fixed structure. Thus, overlaying a relational
structure on top of kernel events provides almost a direct mapping from internal kernel
representations.

Selecting from a kernel event is then semantically equivalent to selecting from a data
stream with a fixed relational definition. For example, the sys_enter_pread64 tracepoint
(with definition in Figure 8) is modeled as a relation with four fields: \(fd\), \(buf\), \(count\), and \(pos\) (which are also the arguments to `pread64` [51]).

```plaintext
name: sys_enter_pread64
ID: 697
format:
   // Four other fields, prefixed common_<field>, exist, but are not
   // available to eBPF programs attached to tracepoints, and are thus
   // omitted.
   field:int __syscall_nr; offset:8; size:4; signed:1;
   field:unsigned int fd; offset:16; size:8; signed:0;
   field:char * buf; offset:24; size:8; signed:0;
   field:size_t count; offset:32; size:8; signed:0;
   field:loff_t pos; offset:40; size:8; signed:0;
```

Figure 8: The `sys_enter_pread64` tracepoint definition, from `/sys/kernel/tracing/events/syscalls/sys_enter_pread64/format`.

However, there are some important caveats to the relational model.

First, an event’s context alone does not fully represent the context available. At each event, the kernel exposes a generic set of system values (e.g. `ktime`, `pid/tgid`, `cgroup`, `cpu`, and `comm`). Thus, in addition to the event-specific context, each event's relational model exposes a fixed set of system attributes.

Second, many events contain complicated kernel structures. The system task struct is a huge, deeply nested, struct; and in block I/O tracepoints, the request struct (Figure 9) contains copious amounts of information, such as the `gendisk`, `block_device`, and more metadata. To represent and access this data, the relation contains a type resembling SQL’s `STRUCT` complex type.

```plaintext
struct request {
    // ... additional fields ...
    struct gendisk *rq_disk;
    struct block_device *part;
    u64 alloc_time_ns;
    u64 start_time_ns;
    short unsigned int wbt_flags;
    // ... additional fields ...
}
```

Figure 9: The struct `request` type at block I/O tracepoints. Access would require nested de-references; for instance, BQL represents accessing the first minor device number of a block I/O request as `request.rq_disk.first_minor`.

Because of these two cases, the event relational model is, in a sense, *denormalized*. Each relation contains attributes that are not specific to them, but rather available at many relations.
every relation contains generic system attributes, and for similar tracepoints (like the block
IO ones), each context contains “duplicate” attribute definitions for identical structs.

Within each event trigger, this denormalization does not inherently impose additional
storage overhead, as each event stream does not literally store all attributes in memory; rather,
on every invocation, the BPF program has read-only access to contexts stored in existing
kernel memory, requiring no additional storage (e.g. each process already has a task struct
associated with it in the kernel, and block IO requests already require a request struct to
be created somewhere in-kernel). However, accessing the normalized fields and transmitting
to user-space does require additional memory and copying to transfer into user-space.

### 3.3.2 Query Output

Each query projects some subset of a kernel event and optionally performs some aggrega-
tions/transformations on the data. To provide a structured representation, the query output is
represented as a `RecordBatch`:

```rust
pub struct RecordBatch {
    pub schema: Arc<Schema>,
    pub records: Vec<Record>,
}
```

Each record batch contains the query’s output schema with a fixed set of `DataType`
s, with a collection of `Record`s representing the actual `DataValue`s of each output record.
RecordBatches are streamed from queries whenever the window steps forward.

The `RecordBatch` structure is intentionally modeled after existing database libraries’
representations, like `sqlx` [61] and Apache Arrow [4].

### 3.4 Query Plans

Given a BQL query, BeeHouse then compiles it into a *query plan* that represents the query
procedure. The conceptual query plan is again inspired heavily by CQL’s design [3]: each
query plan is composed of BQL operators, similar to SQL query plans.

#### 3.4.1 Physical Plan and Query Optimization

The physical implementation of a query plan is split into two components, the *kernel-space*
and *user-space* component.

A key consideration is determining when to emit to user-space, as communicating values
from kernel-space to user-space incurs substantial overhead. Specifically, eBPF programs
emit values to user-space through a concurrent multi-producer single-consumer ring buffer
implemented as a BPF map (`BPF_MAP_TYPE_RINGBUF`), which is consumed by a user-space
process. Much work has been done to optimize the data transmission, with memory-mapped
pages available to user-space applications to avoid kernel-to-user copying, and support for
epoll or busy polling [14] depending on use-case. However, the context switching from
user to kernel space, synchronization and locking overhead of the ringbuf, interrupt request work processing required for polling, and more incur high overhead (we later evaluate these claims in §5.3).

Thus, BeeHouse’s key optimization target is to reduce the total number of events emitted between kernel to user space by pushing computation down into the kernel, under the assumption that kernel-user data transfer’s overhead outweighs the overhead of all other operations (e.g. instruction count, memory access, etc.), similar to how traditional RDBMS implementations seek to reduce total disk IOPS. In the current prototype, BeeHouse’s query planning places any operator that can feasibly be implemented within eBPF’s constraints into kernel space, only deferring to user-space when an operator is not possible in kernel space. Specifically, all operators except for general Joins are implemented purely in kernel space (because eBPF does not have dynamic memory or non-constant-bounded loops, it does not support general Joins).

As part of establishing the user-kernel computation split, BeeHouse performs standard query optimizations (specifically, predicate pushdown, split conjunctive predicates, and projection pushdown [31]). Although eBPF programs do not query external storage—and thus materializing records/attributes does not incur the associated IO cost—BPF programs must still access and copy the attributes to be processed, incurring non-trivial overhead from invoking memory copying and potentially external kernel functions that require synchronization, context switching, and runtime safety checks [56].

BeeHouse also takes advantage of the restriction to tumbling windows to simplify windowing and aggregation logic. In tumbling windows, every window step discards every element in the previous window; thus, windowing needs only to record a constant amount of metadata to determine when to tumble (e.g. items seen for count-based windows, or the start timestamp for time-based windows), and every stateful operator simply clears its synopses, an almost-constant time operation (eBPF does not support clearing maps in kernel-space, so it requires a map iteration to delete each element; however, when values are grouped, the number of group-by keys is often significantly less than the total amount of elements).

Currently, BeeHouse only employs these rule-based optimizations based purely on inferences and assumptions made when processing the query plan. However, recent work attempts to quantify the cost of eBPF operations [56]; thus, future work can explore estimates using data characteristics to perform cost-based optimizations. For instance, if an eBPF operator requires a high amount of costly function invocations or redundant memory allocation in BPF maps, it might be faster to incur the kernel-user data transmission overhead and defer computation to user space.

Operator implementations in user-space follow standard database operator implementations (for instance, a Join could be implemented as a grace hash join or index nested loop join), and are thus elided.
3.5 Code Generation

After processing the query plan, BeeHouse converts the kernel-space operators into eBPF C code.

Due to the constrained set of operators available in BeeHouse, it opts for a simpler approach of composing together multiple operator representations and generating a collection of output header files, with struct, map, and operator definitions in C, included into one source file (details can be found in §4.2).

The generated C code is then dispatched to clang, which compiles it down into eBPF bytecode.

3.6 Query Execution and Post-Processing

BeeHouse then loads the generated eBPF bytecode into the kernel, where the eBPF verifier checks the query for validity, since BeeHouse’s internal catalog should handle logical bindings, and the codegen step should properly adhere to eBPF requirements, users are abstracted away from verifier peculiarities; failure at the verifier indicates a bug in BeeHouse itself, not in the user query.

BeeHouse then processes the loaded eBPF objects, setting global flags and optionally pinning maps (a feature useful for synopsis sharing, but not currently implemented by BeeHouse) before attaching the program to the kernel event. At this point, the generated eBPF program is now running in the kernel, and query output is streamed to the user.
4 Implementation

We implement a BeeHouse prototype with ~3.5k lines of Rust and ~1k lines of C. BeeHouse is compiled as a Rust library, allowing clients to link into its API (defined in Section 3.1).

4.1 BeeHouse Query Parsing

To support parsing of BeeHouse’s BQL syntax, we extend an existing Rust SQL parser, nom-sql, which is based on the nom parser combinator framework [48, 55]. We chose nom-sql over other SQL parsing libraries like sqlparser [60], a top-down operator-precedence (TDOP) parser, due to parser generator frameworks’ general ease of extensibility.

We extend nom-sql with support for kernel event syntax (in standard SQL, slashes—/—are not supported), and BeeHouse-specific operators, such as Window, Histogram, and Quantile, to support streaming semantics and additional analytics.

4.2 Code Generation

Stateless operators are relatively straightforward to generate (for instance, an Equal(a, b) filter becomes if (a == b) {... }) from the logical plan. Since the eBPF stack size is limited to 512 bytes, BeeHouse attempts to consolidate projects, filters, and maps, and avoid storing intermediate state on the stack.

Fully generating stateful operators like aggregations and their associated synopses can become involved; further, the eBPF verifier requires argument types, map definitions, and function invocations to be statically declared, preventing useful generic programming techniques in C, such as function parameters and using void *. Thus, code generation is simplified by representing each stateful operator as a composable template; when a specific query plan is compiled into eBPF code, the template is rendered with the actual values (in some senses, this is similar to monomorphization; the templates represent the generic parameters, while the rendered code is the unique instantiation).

Figure 10 shows an example template for a bpf_for_each_map_elem callback that fetches aggregation values. BeeHouse uses the Handlebars engine [32] to render templates into unique instantiations.

Using these templates, the internal stateful operator implementations can be exposed via a single helper function; thus, in the actual eBPF program code, only that helper function needs to be invoked, greatly simplifying the code generation into a sequence of operators. A generated eBPF program for the query in Figure 7 might then look like Figure 11.

4.3 Prototype Limitations

Our BeeHouse prototype is a proof of concept, and thus has some limitations.

BeeHouse’s query plan analysis is limited, restricting the complexity of generated queries; in particular, nested aggregations, histograms/quantiles, nested selects, post-aggregation
static s64 __get_{{ agg }}_{{ field_name }}_{{ query_name }}_callback(
  struct bpf_map *map,
  group_by_{{ query_name }}_t *key,
  agg_t *agg,
  {{ agg }}_{{ field_name }}_{{ query_name }}_ctx_t *ctx) {
  // Skip if aggregation value is 0
  if (agg->val == 0) {
    return 0;
  }
  // Set agg value (verifier checks are omitted for clarity)
  {{# each group_bys }}
  ctx->buf[ctx->count].{{ field_name }} = key->{{ field_name }};
  {{/ each }}
  ctx->buf[ctx->count].{{ agg }}_{{ field_name }} = agg->val;
  ctx->count += 1;
  return 0;
}

Figure 10: A template for a callback to retrieve aggregated values using bpf_for_each_map_elem.

SEC("tp/syscalls/sys_enter_pread64")
u32 pread_query(struct trace_event_raw_sys_enter* ctx) {
  u64 pid = PID();
  if (pid == 1041370) {
    return 1;
  }
  u64 time = TIME();
  u64 fd = ctx->args[0];
  u64 cpu = CPU();
  u64 count = ctx->args[2];
  bool tumble = window_add(time);
  if (tumble) {
    // window tumbling and emitting to user-space logic...
  }
  insert_count__pread_query({fd, cpu}, 1);
  insert_max_count_pread_query({fd, cpu}, count);
  insert_avg_count_pread_query({fd, cpu}, count);
  return 0;
}

Figure 11: The query in Figure 7, auto-generated using the templates.
processing, and joins are not currently supported. Part of this is by design: due to its context of executing in resource-constrained, performance-sensitive environments, in-kernel queries should incur minimal overhead and avoid unpredictable runtimes that could spike tail latencies (for instance, a large binary join every second could skyrocket tail latencies). Part of it is from verifier restrictions: without support for dynamic memory or unbounded loops, programs must pre-allocate potentially wasteful amounts of memory, or are not feasibly implementable (i.e. joins and arbitrary windows).

Part of its limitations is simply due to the code generation step; the method is rather simple, and does not handle more complicated ASTs yet. Thus, while templated operators enable ease of code generation, complicated control flow amounts to manual checks and code generation. In the future, it would be worth exploring potentially harnessing LLVM IR directly (as bpftrace does) to generate eBPF code, or implementing a more sophisticated compiler.

BeeHouse currently only supports queries that are feasibly implementable in kernel space; queries that require complex post-processing in user-space are not supported.
5 Evaluation

Our evaluation seeks to answer the following questions:

1. What is the abstraction overhead of BeeHouse? How close are BeeHouse-generated queries to the best case per-instance hand-optimized queries? (§5.1)

2. What is the impact of computation pushdown into the kernel on runtime overhead? (§5.2)

3. Where specifically does overhead materialize within an eBPF probe? (§5.3)

All evaluations run on a server machine with two Xeon Gold 6150 CPUs (36× 2.7 GHz) and 377 GiB of RAM. Persistent storage is provided by a Samsung 1TB NVM Drive. The systems runs Ubuntu 22.04 with Linux v5.15.

We evaluate BeeHouse and hand-written queries using a workload derived from real-world observability use-cases. Specifically, a RocksDB application under high load continually executes get commands, reading results from persistent storage. The RocksDB application runs eight writer threads concurrently, stopping once fifty million operations are completed. RocksDB application uniformly randomly selects from pre-generated set of 1,000,000 keys, each containing a data value of 128 bytes, minimizing the effects of caching. A separate process attaches an eBPF program implementing the query in Figure 7 into the kernel at the sys_enter_pread64 tracepoint, and collects and processes the outputted data.

Both applications are pinned to a fixed set of twelve CPUs using taskset, chosen to avoid interference from CPU hyperthreading. We also discuss the effect of restricting the number of cores to eight, such that the RocksDB application then runs within a resource-constrained environment, with worker threads competing with the user-space eBPF management process (in §5.2, this process performs the user-space aggregations).

Before each benchmark, we run the RocksDB operation individually to reduce any disk caching effects.

5.1 End-to-End Evaluation

We first measure the performance of a query from a BeeHouse-generated eBPF program compared to a hand-optimized eBPF program. Ideally, the abstraction overhead of BeeHouse’s higher-level interface and subsequent query processing steps should be minimal, and provide comparable performance to hand-optimized programs with a fraction of the developer overhead. Both programs accumulate values in a BPF map, only computing aggregations and emitting the aggregation results to userspace when the window tumbles (in our setup to implement 7, every second).

We measure the read throughput achieved by RocksDB, first without any eBPF probes attached to determine the baseline expected throughput, then with the BeeHouse-generated eBPF program and the hand-optimized eBPF program. We also measure a wide range of
read latency quantiles to evaluate if either probe has a significant effect on tail latencies. Ideally, the BeeHouse-generated probe should have comparable throughput and quantiles to the hand-optimized probe.

Figure 12 shows the RocksDB read throughput over the three measurements, and figure 13 shows the RocksDB read latencies across various quantiles.

![Figure 12: Throughput comparisons between baseline RocksDB without eBPF probes, with a BeeHouse-generated probe, and with a hand-optimized probe. The hand-optimized probe (1.7% overhead) slightly outperforms the BeeHouse-generated probe (3.2% overhead).](image)

Using eight worker threads across twelve CPUs, RocksDB achieves a baseline read throughput of $\sim 852k$ operations/second without any probes attached. Once the generated probe is attached, the read throughput drops to $\sim 825k$ operations/second. Using the optimized probe, the read throughput is maintained at $\sim 838k$ operations/second. Hence, the BeeHouse-generated probe incurs a 3.2% overhead on reads, compared to the hand-optimized probe’s 1.7% overhead.

The quantile measurements indicate similar results in the median: the baseline RocksDB read latency without an eBPF probe is 8.338μs, versus 8.606μs with the BeeHouse probe (3.2% overhead) and 8.456μs with the hand-optimized probe (1.4% overhead). Moreover, even though both eBPF probes defer aggregation computation and emitting the aggregation results until the window tumbles, the tail latency does not drastically deteriorate with either probe. At the tail (99.9% quantile), the baseline RocksDB read latency is 22.437μs, compared to 23.329μs with the BeeHouse probe (4.0% overhead) and 22.969μs with the hand-optimized probe (2.3% overhead).
5.2 Evaluation of Computation Pushdown into the Kernel

We now measure the impact of pushing computation and our optimization target on performance, comparing a BeeHouse-generated probe to a simple, straightforward eBPF program without query-specific optimizations and in-kernel computations.

We use the program presented in Figure 1 as the baseline. Although simple, the prerequisites to develop an eBPF program like this is non-trivial, requiring knowledge of tracepoint format and available contexts, program types, BPF helper functions, BPF maps, and the ringbuf API [47]. Moreover, developers must be aware of the verifier requirements; omitting the null check (\!q) outputs an error message that requires knowledge of eBPF bytecode to decipher.

In addition, compared to the purely kernel-space processing in the BeeHouse-generated program, the simple program defers all query computation to user-space; specifically, since only the raw record is emitted, the user-space side must handle max/count/average aggregations, groupings, and windowing. This complicates performance benchmarking, as raw throughput no longer accurately indicates program overhead: instead of the query functionality directly impacting throughput (as eBPF programs run in kernel context, on the same process, after a hook point is triggered), a separate process managing the eBPF probe (the “BPF query post-processing process”) handles the vast majority of query functionality.

Thus, for this benchmark we instead measure total kernel clock ticks as a function of work executed across both the RocksDB application and the process managing the eBPF probe. This way, the measurements account for the additional query additional query work, even
if it did not directly affect throughput. For a direct conversion, on our machine, 100 kernel clock ticks is 1 second (this value can be determined with \texttt{sysconf\_\_SC\_CLK\_TCK}). To measure clock ticks, we use \texttt{/proc/<pid>/stat} from procfs for process ticks, and the BPF subsystem itself for BPF program stats (via \texttt{/proc/sys/kernel/bpf\_stats\_enabled}).

We measure five categories of clock ticks: RocksDB user mode, RocksDB kernel mode, BPF probe, BPF post-processing process user mode, and BPF post-processing process kernel mode. From these, we can additionally compute the overhead of the BPF subsystem itself (i.e. the work required to context switch and execute the BPF probe within the kernel) by subtracting the baseline RocksDB kernel mode ticks (the kernel mode ticks for RocksDB-specific work, like \texttt{pread64} syscalls) and the BPF probe ticks (the BPF probe processing cycles) from the RocksDB kernel mode ticks with the BeeHouse/simple probe attached. To limit variance, we run each program fifty times, then compute the average cycle count among all invocations.

![Total CPU Cycles Across RocksDB + eBPF](image)

**Figure 14**: CPU cycles used between RocksDB without eBPF probes, with a BeeHouse-generated probe, and with a simple probe.

**Table 3**: BPF latencies per run.

<table>
<thead>
<tr>
<th>Program</th>
<th>Probe runtime (avg/run)</th>
<th>BPF subsystem overhead (avg/run)</th>
<th>Probe run count</th>
</tr>
</thead>
<tbody>
<tr>
<td>BeeHouse</td>
<td>10.0852s (167.031ns)</td>
<td>7.587s (125.656ns)</td>
<td>60379284</td>
</tr>
<tr>
<td>Simple</td>
<td>26.561s (439.708ns)</td>
<td>7.105s (117.622ns)</td>
<td>60406223</td>
</tr>
</tbody>
</table>

Figure 14 shows the results, and Table 3 contains detailed latency numbers (converted from clock ticks). Across all programs, the user and kernel time for the RocksDB application itself remained roughly equivalent (~ 41450 ticks). Likewise, the overhead from the BPF subsystem itself was relatively consistent across programs (~ 120ns per BPF probe invocation).
Figure 15: End-to-end read throughput comparison on twelve (a) and eight (b) CPUs. With twelve CPUs, the simple BPF probe incurs $\sim 7.2\%$ overhead; with eight CPUs (i.e. the RocksDB and query post-processing processes compete for cores), the simple BPF probe’s overhead spikes to $\sim 17.6\%$. In both instances, because the bulk of the BeeHouse-generated probe’s work is done in-kernel, changes in the core count do not impact the overhead ($\sim 3.2\%$).

Perhaps surprisingly, despite performing less computation in-kernel, the simple BPF program actually incurs more overhead per BPF probe execution. Further, the BPF query post-processing process must now also devote significant CPU time towards computing the aggregations in user-space and polling data from kernel-space. The key difference lies in data transfer from kernel to user space: although the simple BPF program performs less computation in-kernel, copying data from kernel to user space incurs significantly more overhead than all other kernel computation. We investigate this below, in §5.3.

To further investigate, we evaluate each probe end-to-end, measuring RocksDB read throughput when both processes are run on 12 CPUs as before, then on 8 CPUs to simulate a resource-constrained environment.

Figure 15 shows the results. From the clock tick evaluation before, the twelve CPUs end-to-end evaluation is expected: the BeeHouse probe outperforms the simple eBPF probe with less overhead (3.2\% vs 7.2\%) by prioritizing a more optimized in-kernel computation approach. Further, when constrained to eight CPUs (and thus the RocksDB process and the query post-processing process compete for CPU cycles), the simple eBPF probe’s performance further deteriorates (17.6\% overhead) while the BeeHouse probe’s—which prioritizes in-kernel computation on the same core as the RocksDB process itself, preventing any competition—performance remains the same.

5.3 Performance Drilldown

We briefly investigate the cause behind the simple probe’s high overhead.
Figure 16: Flamegraphs of RocksDB’s `pread64` syscall under the BeeHouse probe (a) and simple probe (b). In the BeeHouse probe, the BPF probe occupies only a fraction of total runtime in the `pread64` syscall: CPU cycles are distributed similarly among BPF helper functions. In contrast, in the simple probe, the BPF probe occupies over half of `pread64` total runtime, with ringbuf reserves/submits to userspace using the majority of CPU cycles.

Figure 16 shows two flamegraphs (generated using cargo `flamegraph`) comparing RocksDB’s `do_syscall_64` stack trace and rough execution proportions under a BeeHouse probe, and the simple probe. In the simple probe, ringbuf reserves/submits to userspace occupy more than half the `pread64` syscall’s runtime, and a majority of the BPF probe’s work. In contrast, the BeeHouse probe—which aims to replicate the optimized probe’s logic—spends only a fraction of the `pread64` syscall’s runtime on the BPF probe, with CPU cycles distributed similarly among various BPF helper functions (specifically, `bpf_ktime_get_ns()` and `bpf_map_lookup_elem`).

This also shows the BPF subsystem’s overhead: in both programs, entering the BPF program from the syscall context and exiting the BPF program into user mode incur a non-trivial penalty (which Table 3 shows); in the BeeHouse-generated probe, this overhead roughly equals the BPF probe’s overhead itself.

5.4 Discussion

BeeHouse’s probe itself, despite performing aggregations every time the window tumbles, manages to maintain a low tail latency. On average, the probe runs for ~160ns, with around ~120ns spent on infrastructure supporting the hook.

While BeeHouse is not a complete zero-cost abstraction, it incurs a low additional overhead over the hand-optimized eBPF program, and provides significant performance improvements over a simple eBPF probe that a resource-constrained environment further exacerbates. Further, in most cases, the performance gap between BeeHouse and hand-
optimized code is closeable. For instance, the key optimization the hand-optimized probe employs that BeeHouse does not is per-CPU maps, since one of the group-by keys was CPU. If BeeHouse’s query optimizer can identify special cases to use per-CPU or task-local storage, it is likely that BeeHouse probes can become on-par with hand-optimized code, with a fraction of the development effort.
6 Future Work

BeeHouse is still a prototype; there is much future work left to explore.

We have shown that performing as much aggregation and filtering in kernel space as possible significantly lowers overhead by reducing the amount of data transmitted between to user space (§5.3). Cost-based analysis and optimizations provide an opportunity to further develop this research: as recent work starts to quantify performance characteristics [45] and latencies of specific eBPF routines [56], it would be interesting to produce a cost estimate based not only on data transfer cost, but also specific eBPF routines (e.g. kernel memory accesses, hash map iterations, or helper function invocations). Further, since eBPF programs execute continuously, BeeHouse could potentially gather statistics on data characteristics and use runtime flags to dynamically toggle operators.

These cost-based optimizations become increasingly important as the Linux kernel supports more features and gradually removes eBPF restrictions. For instance, Linux v5.17 introduces an arbitrary bpf_loop that relaxes loop restrictions [37], and Linux 6+ introduces kfuncs and non-BPF-map based data structures like linked lists and red-black trees, opening the floor for performant dynamic memory [13, 42]. With these constructs, joins can become feasible in kernel space, and so cost analysis to minimize tail latencies becomes even more pertinent.

BeeHouse also adopts a relatively simple streaming approach that contains opportunities for sophistication. Although eBPF functions are event-based and thus cannot be manually scheduled, queries could potentially share synopses (e.g. if separate developers are querying the same tracepoint), and various probabilistic sketch algorithms for efficient sub-linear approximations might reduce memory/CPU requirements. For instance, some prior work [45] created an implementation of the count-min sketch; it would be interesting to investigate the performance and feasibility of other sketches, like HyperLogLog, Theta Sketches, and the t-digest/q-digest for quantiles.
7 Conclusion

BeeHouse is an eBPF query engine that facilitates performant kernel data collection via a high level interface resembling SQL, exposing a relational layer over complex kernel tracing infrastructure. BeeHouse eases the burden of understanding eBPF internals and developing custom eBPF programs, making low-overhead kernel data collection accessible to application developers.

We evaluate BeeHouse to determine its abstraction overhead, and find that it incurs some additional—but resolvable—overhead compared to an ideal, hand-optimized probe. We find that BeeHouse’s optimization target of reducing the amount of data transferred from kernel to user space is an effective one, seeing up to $6 \times$ improvement over a simple eBPF probe without query-specific optimizations that emits all data to user-space before computing.
References


[26] eBPF Documentation. URL: https://ebpf.io/what-is-ebpf/ (visited on 05/06/2024).

[27] eBPF Verifier. URL: https://docs.kernel.org/bpf/verifier.html.


[37] Joanne Koong. Add bpf_loop helper. URL: https://lore.kernel.org/bpf/87tuft7ff7.fsf@toke.dk/T/.

[38] Ling Liu, C. Pu, and Wei Tang. “Continual queries for Internet scale event-driven information delivery”. In: IEEE Transactions on Knowledge and Data Engineering 11.4 (1999), pages 610–628.


40
[42] Dave Marchevsky. BPF rbtree next-gen datastructure. URL: https://lwn.net/ml/bpf/20230209174144.3280955-1-davemarchevsky@fb.com/.


[51] pread64 manpage. URL: https://linux.die.net/man/2/pread64.

[52] Alastair Robertson and Brandon Gregg. bpftrace. URL: https://github.com/bpftrace/bpftrace.


[60] SQL Parser for Rust. URL: https://docs.rs/sqlparser/latest/sqlparser/.

[61] sqlx. URL: https://docs.rs/sqlx/latest/sqlx/.


[64] SystemTap. URL: https://sourceware.org/systemtap/ (visited on 05/06/2024).


A Code Artifacts

The code for BeeHouse can be found at https://github.com/ringtack/ebql, and the benchmark code can be found at https://github.com/ringtack/ebql-benchmarks (in particular, this contains the optimized code used in §5.1).