BPF-DB: A Kernel-Embedded Transactional Database Management System For eBPF Applications

MATTHEW BUTROVICH, SAMUEL ARCH, WAN SHEN LIM, WILLIAM ZHANG, JIGNESH M. PATEL, and ANDREW PAVLO, Carnegie Mellon University, USA

Developers rely on the eBPF framework to augment operating system (OS) behavior for the betterment of database management system (DBMS) without having to modify kernel code. But eBPF's verifier limits program complexity and data management functionality. As a result eBPF's storage options are limited to kernel-resident, non-durable data structures that lack transactional guarantees.

Inspired by embedded DBMSs for user-space applications, this paper present BPF-DB, an OS-embedded DBMS that offers transactional data management for eBPF applications. We explore the storage management and concurrency control challenges associated with DBMS design in eBPF's restrictive execution environment. We demonstrate BPF-DB's capabilities with two applications based on real-world systems. The first is a Redis-compatible in-memory DBMS that uses BPF-DB as its transactional storage engine. This system matches the performance of state-of-the-art implementations while offering stronger transactional guarantees. The second application implements a stored procedure-based DBMS that provides serializable multi-statement transactions. We compare this application against VoltDB, with BPF-DB achieving 43% higher throughput. BPF-DB's robust and high-performance transactional semantics enable emerging kernel-space applications.

CCS Concepts: • **Information systems → Data management systems**.

Additional Key Words and Phrases: eBPF; embedded databases; operating system extensibility

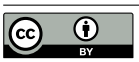
ACM Reference Format:
Matthew Butrovich, Samuel Arch, Wan Shen Lim, William Zhang, Jignesh M. Patel, and Andrew Pavlo. 2025. BPF-DB: A Kernel-Embedded Transactional Database Management System For eBPF Applications. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 135 (June 2025), 27 pages. https://doi.org/10.1145/3725272

1 Introduction

Cloud services increasingly rely on eBPF to solve large-scale networking, security, observability, and performance problems [10, 12, 14]. For example, Meta loads over 40 eBPF programs on every server, with hundreds more loaded on demand [63]. eBPF enables *user-bypass* methods that push application logic into the operating system (OS) without modifying kernel code [28]. Embedding custom behavior into OS services improves performance by reducing user-space application overheads like process scheduling, system calls, and data movement between kernel-space.

As more developers adopt eBPF and apply user-bypass techniques, there is a need for more robust data management solutions. The only choice for programs today is to store their data in *eBPF maps* (e.g., hash tables) that do not offer ACID (i.e., atomicity, consistency, isolation, durability) properties. For example, eBPF offers safe concurrent accesses for multi-threaded applications based on Linux's read-copy-update (RCU) semantics, but only for a single read/update operation. This limitation forces developers to implement their own synchronization mechanisms to avoid race conditions.

Authors' Contact Information: Matthew Butrovich, mbutrovi@cs.cmu.edu; Samuel Arch, sarch@cs.cmu.edu; Wan Shen Lim, wanshenl@cs.cmu.edu; William Zhang, wz2@cs.cmu.edu; Jignesh M. Patel, jignesh@cmu.edu; Andrew Pavlo, pavlo@cs.cmu.edu, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA.



ACM 2836-6573/2025/6-ART135
https://doi.org/10.1145/3725272

Similarly, eBPF programs do not support inter-process communication (IPC) and must coordinate concurrent execution through eBPF maps with bespoke logic that is onerous and error-prone. Lastly, eBPF maps reside in kernel-space memory without a method to persist their contents on disk.

As a workaround, eBPF developers may choose to store their application state in a user-space DBMS. For example, Meta's eBPF-based load-balancer bundles both SQLite and RocksDB as dependencies for state management [16]. However, eBPF programs cannot communicate synchronously with user-space processes. This restriction makes it impossible for event-driven, time-limited eBPF programs to retrieve and modify data within their execution budget.

A better approach is for developers to create user-bypass applications using a eBPF-native DBMS that provides the critical features but without sacrificing their system's security or stability. However, eBPF's *verifier* restricts using an off-the-shelf embedded DBMS because it prevents linking outside libraries. Furthermore, existing DBMSs do not conform to the verifier's rules for safe memory access, termination, and limited API. Refactoring an existing DBMS to conform to eBPF's verifier is impractical—instead a DBMS for eBPF must be designed from scratch.

Given this, we present **BPF-DB**, a kernel-embedded DBMS that offers ACID-compliant transactions for eBPF applications. We discuss our deviations from traditional DBMS design to suit eBPF's restrictive execution environment. To demonstrate BPF-DB, we build two distinct, kernel-resident applications. The first is a Redis-compatible server that we compare against state-of-the-art systems. The second system demonstrates serializable transactions running in kernel-space. Our experimental analysis shows that BPF-DB achieves up to 43% higher throughput than existing systems while supporting multi-statement transactions. We contend that BPF-DB's performance with support ACID-compliant transactions will enable new opportunities for kernel-space applications built for OS-embedded environments like eBPF.

This paper is organized as follows. We first discuss embedded DBMSs and expand on the limited data management options in eBPF's runtime in Section 2. Then, we introduce BPF-DB in Section 3. Next, in Section 4, we detail the implementation challenges and design decisions of BPF-DB in the context of traditional DBMS solutions. We demonstrate two applications built on BPF-DB in Section 5 and then evaluate them in Section 6. We survey related work in Section 7, briefly discuss future opportunities in Section 8, and provide concluding remarks in Section 9.

2 Background

McCanne and Jacobson introduced the original Berkeley Packet Filter (BPF) library in 1993 [49]. BPF enabled the execution of user-written code in the OS kernel via a reduced instruction set virtual machine (VM). With only 22 instructions and minimal capabilities, developers wrote BPF programs as bytecode rather than compiling from a higher-level language.

Two decades later, Linux introduced "extended" BPF (eBPF) that expands its semantics to enable developers to create safe, event-driven programs running in kernel-space without the complexity or safety concerns of kernel modules [62]. eBPF provides a more robust VM than the original BPF VM with three key features: (1) kernel-embedded data structures (i.e., eBPF maps), (2) access to kernel functions via eBPF helpers, and (3) greater expressiveness via increased program length and additional registers. Microsoft has an open-source, clean room implementation of eBPF for Windows, but it currently does not support all of Linux's features [14]. Similarly, Android recently added limited eBPF support [8].

eBPF programs still use a limited instruction set, though their capabilities have increased to include function calls, loops, and the ability to tail call from one eBPF program to others. Rather than writing bytecode, modern GCC and LLVM compilers enable developers to write eBPF programs in higher-level languages (e.g., C, Rust, Go). eBPF programs load into the kernel via the `bpf()` system

Fig. 1. eBPF Program Using eBPF Maps – The kernel offers non-durable, in-memory storage for eBPF programs called maps.

call, which first verifies the program before just-in-time (JIT) compiling the bytecode to native machine code for faster runtime execution.

eBPF program execution is event-driven. After loading their eBPF programs into kernel-space, developers then associate corresponding events (e.g., functions, static tracepoints) to trigger program execution. When a running process hits the attachment point, the eBPF program starts execution in privileged mode.

2.1 eBPF Verifier

Because the CPU is in privileged mode when eBPF programs run, the kernel requires them to pass a verification step before loading. eBPF's *verifier* enforces kernel API compliance, memory access safety, execution bounds, and instruction count. The verifier generates a *control flow graph* (CFG) for all possible branches of the program and enforces limits like 512 B maximum stack size.

While exploring the CFG, eBPF's verifier terminates after processing 1 M instructions and rejects the program. Although this creates a *de facto* limit of 1 M instructions for an eBPF program, the limit is much lower in practice due to how the verifier processes programs' conditional logic. The verifier explores all branch and loop states, exponentially increasing verifier work with nested branches or loops. For example, if a loop examines eight elements of an array, the verifier simulates the program state through all loop iterations. Any eBPF code inside the loop will count eight times toward the verifier's 1 M instruction limit [2].

2.2 eBPF Maps

To overcome this limit, eBPF supports *tail calls* between eBPF programs. Unlike calling a function, tail calling jumps to the new program location, overwrites the call stack, and does not return to the caller upon completion. For example, one eBPF program could parse a network buffer and then tail call into different eBPF programs depending on the buffer's contents. Decomposing application logic into separate eBPF programs and chaining them with tail calls allows applications to expand the verifier's instruction limits. However, the verifier restricts programs to a maximum of 33 tail calls in a sequence, imposing a total verification complexity limit of 34 M eBPF instructions across 34 programs. Because the stack does not persist between tail calls, applications must rely on eBPF maps to carry state between eBPF programs.

eBPF programs store data using a a key-value interface using in-memory, non-durable data structures called *maps* that reside in kernel-space. Figure 1 shows an example of a program in kernel-space using eBPF helpers to interact with maps. The underlying behavior, supported data types, and storage structures vary per map type. Some maps support custom key types (e.g., hash table), and others only expect numeric indexes for their keys (e.g., array).

Although maps are safe for concurrent access due to the Linux's read-copy-update (RCU) interface, each access is a single atomic operation [50]. These abstractions are sufficient if programs attach to points without concurrent execution (e.g., single-threaded application). eBPF does not provide atomic read and write operations on multiple entries either within the same map or across multiple maps. Since multiple programs can attach to multi-threaded applications or launch from kernel tasks via CPU interrupts, developers must consider race conditions when interleaving map operations.

Synchronization of eBPF programs is challenging to implement without a coordination mechanism and access to the kernel's RCU primitives. eBPF maps do not expose any way to roll back operations, so guaranteeing execution with "exactly once" semantics is non-obvious. As a reduced instruction set, eBPF supports a subset of common atomic instructions (e.g., `ADD`, `AND`, `OR`, `XOR`, `XCHG`, `CMPXCHG`) [7]. The kernel provides mutual exclusion via a spin latch eBPF helper, but the verifier restricts the scenarios when programs can use it. For example, the verifier enforces that the eBPF program does not call functions while holding a latch, and programs cannot hold more than one latch at a time to prevent deadlocks. As such, programs cannot use a spin latch to provide mutual exclusion to access multiple entries simultaneously.

2.3 eBPF Applications

The need for eBPF grew out of high-performance network packet processing, but its applications now extend to container coordination, performance engineering, security enforcement, and more. Cilium is a container networking interface (CNI) encompassing all these aspects, allowing developers to customize their Kubernetes container environment in kernel-space [10]. Meta drives a significant amount of Linux kernel enhancement for eBPF, and their Katran load balancer manages traffic in all their data centers [12].

Observability has been a significant driver for eBPF adoption due to its flexibility and native integration with the Linux kernel [38]. Off-the-shelf tools like BCC bundle eBPF programs to measure resources like I/O activity, CPU time, and file system behavior [13]. For application-specific tracing, bpftrace provides a high-level scripting language that generates eBPF programs at runtime, allowing developers to analyze system performance without downtime [9]. Combining these approaches offers developers new ways to debug, test, and optimize system software [51].

Due to its safe integration with the Linux kernel, eBPF presents new ways for DBMSs to tune OS-specific knobs. In 2023, Oracle introduced bpftune to automatically adjust network stack parameters with the eventual goal of extending its behavior to other knobs [47]. Furthermore, researchers recently explored dynamically adjusting OS mutex policies at runtime using eBPF [54, 55].

Enterprises continually create more complex and critical applications based on eBPF. For example, Microsoft recently deployed their eBPF-based endpoint security for Microsoft Defender and deprecated the old kernel access methods [4]. eBPF provides safer and more efficient ways to modify kernel behavior for security applications without relying on hazardous kernel drivers [52]. However, as application complexity grows, developers increasingly describe the challenges of maintaining program state in the simple data structures provided by eBPF maps [73].

2.4 Embedded DBMSs

Most DBMSs run as dedicated user-space processes where client applications send queries and retrieve results from the DBMS via a DBMS-specific inter-process communication (IPC) protocol. In contrast, *embedded DBMSs* run in the same process as the client application, either by including the DBMS source code or linking the DBMS as a library. Clients interact with an embedded DBMS through an API to access database contents in the context of their own source code, thread execution, and process address space. Developers may use an embedded DBMS if their application is the only database client or the deployment environment does not offer robust multitasking or IPC (e.g., embedded systems).

Embedded DBMSs vary in their capabilities and complexity. SQL systems like SQLite and DuckDB offer ACID transactions and robust analytics performance, respectively [33, 56]. Other embedded DBMSs like Berkeley DB and RocksDB present a key-value interface and enable client applications to run transactions over database contents [31, 53]. Embedded DBMSs simplify the non-trivial

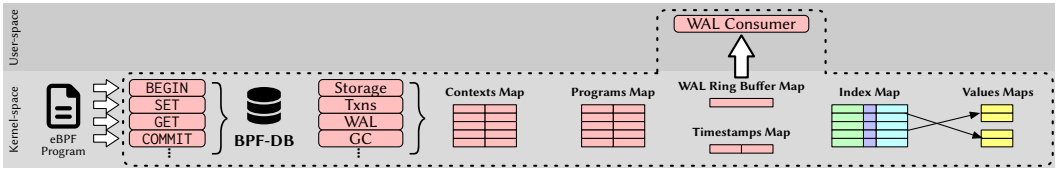


Fig. 2. eBPF Program Using BPF-DB – eBPF programs store their data in kernel-resident, in-memory data structures called eBPF maps. BPF-DB provides robust data management for eBPF programs with serializable transactions and optional durability with write-ahead logging (WAL).

work of query processing and data integrity so developers can focus on their application's core functionalities.

There are two common patterns for building applications with embedded DBMSs. The first approach is where the application bundles an embedded DBMS to manage its state. The second approach is a specialization of the first, where the client application is a full-featured DBMS that uses the embedded DBMS as its transactional storage manager. For example, RocksDB serves as the storage manager for MySQL and Cassandra variants MyRocks and Rocksandra, respectively [1, 48].

To our knowledge, there are no other attempts at designing an in-process, kernel-embedded, transactional DBMS for eBPF programs. eBPF's verifier prevents using existing embedded DBMSs because it does not allow linking outside libraries. Furthermore, existing DBMSs do not conform to the verifier's rules for safe memory access, termination, and limited API. Refactoring an existing DBMS to conform to eBPF's verifier is impractical. We contend that a new DBMS architecture that is tailored to eBPF's environment is necessary.

3 BPF-DB Overview

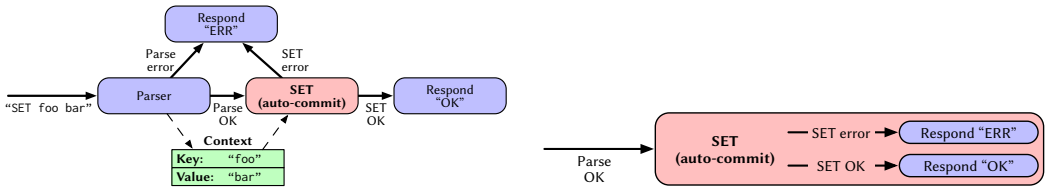
Given the above challenges, we present a DBMS architecture that provides transactional data management for eBPF programs. **BPF-DB** is an in-memory DBMS that runs in kernel-space and supports serializable ACID transactions via a key-value interface. Figure 1 shows an eBPF program storing data in non-durable eBPF maps using kernel-provided helper functions. In contrast, Figure 2 presents an eBPF program using BPF-DB's *operators* (e.g., `BEGIN`, `GET`, `COMMIT`) to manage data with ACID guarantees. BPF-DB stores database contents in eBPF maps that it optionally persists via write-ahead logging (WAL). Client applications only access the database through BPF-DB's operators, but they can also use other eBPF maps to store ephemeral data.

To deploy BPF-DB, eBPF developers implement their application logic in standalone programs. Developers then compile their application with BPF-DB and load the eBPF programs into the kernel with the `bpf()` system call. Once verified and loaded into the kernel, BPF-DB instantiates its eBPF maps, and applications can start accessing database contents. We next describe the ACID properties that BPF-DB provides for eBPF data. Then, we discuss the programming model for accessing operators. Lastly, we detail BPF-DB's operator behavior.

3.1 ACID Semantics

Transactions enable developers to group multiple database accesses together and appear as though they occur as a single logical event. BPF-DB also provides roll-back semantics to eBPF programs. When combined with transaction atomicity, BPF-DB ensures that either all operations in a transaction complete or none of them complete.

BPF-DB makes it easier for eBPF developers to consider race conditions through consistency and isolation. For example, BPF-DB's key-value API maintains primary key semantics so applications can uniquely address records and use BPF-DB as their primary data store. For eBPF programs accessing BPF-DB from multiple threads, their transactions proceed as if running in isolation as



(a) SET operator with tail call continuations **(b)** SET operator with inline continuations

Fig. 3. Continuations for SET – BPF-DB's SET operator (shown in red) with application logic (shown in blue). Continuation complexity determines whether they are tail called programs or logic embedded in the operator.

long as their operations do not violate serializability. Developers receive feedback when their transactions cannot commit and can react depending on the semantics of their application (e.g., retry). BPF-DB's design provides eBPF programs with "exactly once" semantics. Lastly, BPF-DB can optionally record modifying transactions to a write-ahead log (WAL), ensuring the durability of all writes to the DBMS. eBPF maps are not persistent and do not provide an interface to save their contents to disk. BPF-DB addresses this shortcoming by exporting a WAL of its operations that a user-space application can store on disk or replicate over the network.

3.2 Continuation-Passing Style

Developers define eBPF program sequences akin to stored procedures that interleave application logic with BPF-DB's operators. However, BPF-DB's operators are not functions and instead are self-contained eBPF programs that applications tail call using *continuation-passing style* (CPS) logic [65]. With this form of control flow, developers define continuations when tail calling an operator. Upon completion, rather than returning to its caller as in direct style, the operator calls the appropriate continuation. Developers construct procedures that tail call through multiple eBPF programs and BPF-DB operators through continuations.

Depending on the operator, developers define continuations from four possible states. `GET` operators require continuations for whether a key exists. The OK state signifies that the called operator succeeded and is ready to tail call its continuation. In contrast, the Error state denotes that the request cannot proceed and that the transaction is now in a failed state and cannot commit. Error states can originate from several sources. First, BPF-DB employs locking to provide ACID semantics, and lock conflicts for a `GET` or `SET` operator cause an Error state. Second, if WAL is enabled and BPF-DB cannot write an entry due to a full buffer, it will generate an Error state. Lastly, BPF-DB checks the return code of the kernel's eBPF helper function calls. If any of these functions fail due to an internal constraint (e.g., out of memory), then BPF-DB reports an Error state.

In an Error scenario, BPF-DB operators perform any local cleanup before calling continuations so as not to leave the system in an invalid state. For example, if BPF-DB's `SET` fails to write a value after already acquiring the lock on a key, it reverses the locking process, leaves the key metadata unchanged, reports an Error state, and calls the appropriate continuation. BPF-DB does not automatically roll back entire transactions in an Error state. Instead, it gives applications the opportunity to perform additional processing (e.g., clean up state, process remaining events) before applications explicitly call the `ROLL BACK` operator.

Figure 3 shows CFGs of BPF-DB's CPS design integrated with custom application logic. In Figure 3a, each box represents an eBPF program, with application logic in blue and BPF-DB operators in red. In this example, a parser written in eBPF receives a query to set the key "foo" to the value "bar". If the string is a valid query, the parser populates a Context object that BPF-DB

Table 1. BPF-DB Operators and Continuation States – Developers define logic for each BPF-DB operator's continuations.

Operator	Continuation States
`BEGIN`	(1) OK, (2) Error
`BEGIN` (read-only)	(1) OK
`GET`	(1) Key found, (2) Key not found, (3) Error
`GET` (read-only)	(1) Key found, (2) Key not found
`GET` (auto-commit)	(1) Key found, (2) Key not found
`SET`	(1) OK, (2) Error
`SET` (auto-commit)	(1) OK, (2) Error
`COMMIT`	(1) OK, (2) Error
`COMMIT` (read-only)	(1) OK
`ROLL BACK`	(1) OK

stores in an eBPF map with both the key and the value and tail calls into the `SET` operator. The `SET` operator runs and either tail calls to the error handler because it could not set the value or tail calls to the success handler. If the continuations' logic is simple enough not to add significant verifier complexity, the code may be inlined with the operator, as shown in Figure 3b. This optimization improves performance by reducing the number of tail calls between eBPF programs that otherwise incur an overhead of tens of nanoseconds [29].

BPF-DB's CPS architecture provides multiple benefits. First, its operators are self-contained eBPF programs accessed through tail calls, so their eBPF instructions do not count against a verifier's instruction limit. Second, we define BPF-DB's operators (along with their continuations) separately so developers can write applications in any eBPF-supported programming language and then compile and load BPF-DB into the Linux kernel independently. Lastly, the separation of BPF-DB and application logic simplifies development and provides runtime benefits. The Linux kernel atomically loads and unloads eBPF programs so developers can independently modify applications and BPF-DB logic without downtime.

3.3 Operators

BPF-DB presents a key-value interface to applications. Table 1 shows its operators along with their possible tail call continuations. Application must define all continuations for operators they use. Otherwise, BPF-DB's code will fail to compile. Like other key-value DBMSs, BPF-DB's operators are not strongly typed. Keys and values are byte sequences with corresponding length attributes, and applications use their own data encoding scheme.

BPF-DB provides explicit read-only transactions for performance optimization and developer convenience. Because BPF-DB maintains multiple versions of database records, non-modifying transactions can proceed without blocking writers. Read-only transactions do not contain any Error states because they do not acquire locks or flush log records. Read-only transactions also do not perform writes on database contents using eBPF helpers. This design improves BPF-DB's performance and reduces the number of continuations for application developers to define.

A benefit of compiling BPF-DB operators with client applications is that compilers can optimize the generated eBPF code for continuation definitions and knob settings (e.g., WAL enabled, maximum number of versions, maximum locks per transaction). The result is specialized operators akin to pre-compiled stored procedures. For example, turning logging on or off is a compile-time constant for BPF-DB operators. If logging is off at compilation time, an optimizing compiler like Clang or GCC will omit all WAL logic during dead code elimination. However, this design choice does

not prevent developers from enabling BPF-DB's WAL during runtime. By recompiling BPF-DB's operators with logging enabled, the compiler generates new eBPF programs that can be atomically swapped with existing programs using the `bpf()` system call.

Since BPF-DB exposes a low-level API (similar to RocksDB), developers are responsible for producing valid BPF-DB procedures. For example, custom procedures must not `BEGIN` a modifying transaction and then call BPF-DB's read-only `COMMIT` operator. This is a suitable design choice for multiple reasons. First, eBPF is amenable to code generation from high-level scripting languages. A future domain-specific language and compiler could enforce BPF-DB constraints at code generation time, similar to checks within eBPF's verifier. We discuss the concept of code generation in greater depth in Section 8. Second, if an application provides interactive transactions to clients, whereby clients invoke behavior that violates BPF-DB's semantics, the application can implement runtime validation logic. For example, the parser in Figure 3 could handle a client calling an invalid command sequence with its own continuations rather than calling into BPF-DB. Third, eBPF developers are used to disciplined software design due to the eBPF verifier's strict requirements, and BPF-DB's requirements are no more onerous than existing eBPF limitations.

4 System Architecture

As described in Section 3, BPF-DB is an embedded DBMS that stores database contents in eBPF maps with ACID guarantees. BPF-DB uses several eBPF maps to coordinate global state in the system, as shown in Figure 2. The `contexts` map stores context objects for each client session. Each context contains BPF-DB metadata like the current transaction status, transaction timestamp, locks held, and buffers for I/O with operators (i.e., key, value). Developers can also append custom fields to the context object to carry application-specific state between tail calls.

User-space DBMSs often map a single process or thread to a session and use this as its identifier, however BPF-DB cannot make any assumptions about its operators' execution contexts. eBPF programs attach arbitrarily, so applications must define their own semantics for a session identifier. For example, an application attached in the network layer cannot use process ID as a stable session identifier because network sessions can migrate to different handlers depending on system load. In this case, a combination of IP addresses and ports can reliably map to a distinct client session.

The `timestamps` map maintains a single struct that all sessions use. We detail BPF-DB's use of these values in Sections 4.1 and 4.2. The `programs` map contains all of the application's eBPF programs, including client logic and BPF-DB operators. Developers use this map to define continuations that tail call eBPF programs, as described in Section 3.2.

4.1 Storage Management

BPF-DB stores a database in kernel-resident data structures to access and manage the contents for other eBPF applications. As detailed in Section 2.2, the only memory that eBPF programs can use to retain data between program executions is eBPF maps. In this execution environment, there is no API to access the heap (i.e., dynamic memory allocation) like `malloc()`.

Given this, BPF-DB uses eBPF maps to store its database contents. eBPF programs must define their eBPF maps and corresponding key and value sizes before being loaded into the kernel. This requirement exists because the verifier must ensure programs do not access keys or values out of bounds. Because BPF-DB stores its data in kernel memory independent from its eBPF programs, developers can load and unload client applications and BPF-DB operators without losing database contents. This design is similar to how Scuba achieves system restarts without reloading data by storing its database contents in shared system memory [35].

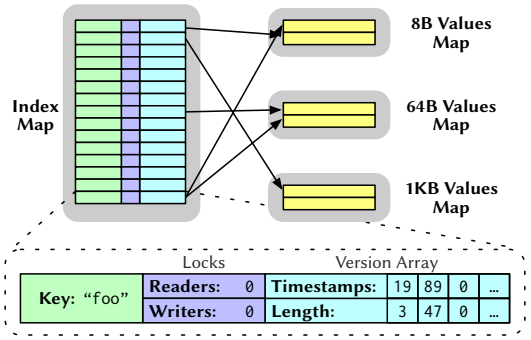


Fig. 4. BPF-DB's Index and Value Maps – BPF-DB uses a top-level index that stores locks and version metadata. BPF-DB stores values in separate eBPF maps based on their size class. The zoomed inset shows an index entry that maps to multiple values based on their commit timestamps.

A naïve implementation could use a single eBPF map for the entire database, where the DBMS restricts keys and values to one size. However, this design trades simplicity for memory fragmentation: every key-value pair uses the maximum number of bytes regardless of size. BPF-DB works around this lack of dynamism by instantiating eBPF maps for different size classes. Then, BPF-DB stores values in the eBPF map that best suits their sizes, similar to user-space memory allocators that maintain free lists for different object sizes [25, 26, 39, 70].

Figure 4 shows an overview of BPF-DB's hierarchical eBPF maps that store its database contents. The first level is the database *index* that contains keys and metadata (e.g., locks, timestamps) for every record. BPF-DB always performs a lookup in the database index first to access a record. For a `GET` operation, the absence of an index entry is enough information to determine that a key does not exist. If an index entry does exist, then the `GET` operation performs a second eBPF map lookup on the correct value map.

BPF-DB employs multi-versioned concurrency control (MVCC) to provide isolation between concurrent transactions. The zoomed inset of Figure 4 details the contents of a single entry in the database index, showing a key "foo" with two corresponding versions: (1) a value created at timestamp 19 with a size of three bytes, and (2) a newer value created at timestamp 89 in a different size class. Rather than using a centralized lock table, BPF-DB stores locks inline with the key and version metadata [37]. We elaborate on BPF-DB's locking and concurrency control in Section 4.2.

Linux restricts total eBPF map memory to the same limits as other kernel data. A combination of kernel compile-time knobs and physical memory configuration determine the exact memory values, though there are mechanisms to manage eBPF memory usage. For example, prior to Linux kernel v5.11, `rlimit` controlled eBPF memory limits, whereas more recent versions expose cgroup configurations for finer control [40].

Each database index entry also contains MVCC information for its key. Unlike most MVCC DBMSs, the number of versions for each entry in BPF-DB is bounded. There are several reasons for this design choice. (1) The eBPF verifier requires compile-time bounds on all loops, so traversing a version chain requires an *a priori* limit on the number of elements a eBPF program will inspect. (2) The lack of dynamic memory allocation precludes a linked list implementation for the version chain. (3) BPF-DB assumes short transactions; thus, it will not need to maintain many versions of an entry for extended periods.

Most MVCC DBMSs also order their version chains based on timestamps, either from newest-to-oldest or oldest-to-newest [71]. The former optimizes for recent transactions to scan fewer chain entries at the expense of increased contention and updating indexes. The latter simplifies

index maintenance during updates, deferring their updates until garbage collection (GC) prunes old versions.

BPF-DB implements neither of these version chain orderings—instead, it maintains an unordered array of 8 B version timestamps. When a `GET` or `SET` operator needs to determine the correct version of a record to access based on its timestamp, it scans the whole array. As described above, version chains must be bounded to satisfy the eBPF verifier, and BPF-DB assumes transactions are short-lived and thus does not need to maintain many old versions. For these reasons, we assume that a version array fits within a CPU's 64 B L1 cache line, corresponding to a maximum number of eight versions. Scanning a data structure that fits within a cache line is fast, and the added logic of maintaining an ordered version array is not worth the verifier complexity penalty.

BPF-DB performs GC to prune old versions that are not visible to any active transaction. BPF-DB employs a cooperative GC approach where `SET` operators, if the version array is full, first remove obsolete versions [44]. If `SET` is unable to prune any versions (i.e., all versions are still visible to some running transaction) then it reports an Error state. BPF-DB employs cooperative GC to exploit the fact that the `SET` operator must acquire the write lock on an entry to update it; this approach avoids added contention from a separate GC worker acquiring locks. This design has the trade-off of potentially allowing stale versions to persist in the database until a key's version array fills and triggers cooperative GC. If cooperative GC retains too many stale versions, a future optimization is to create a background kernel worker that uses the `bpf_timer()` helper to perform GC at a fixed interval. This approach would come at the expense of increased per-entry lock and eBPF map latch contention. This design is evocative of Deuteronomy, which applies soft and hard limits to the total number of versions in the system to trigger GC [46]. However, BPF-DB's limits are more fine-grained on a per-key basis. We explore changing version array sizes and disabling MVCC in Sections 6.8 and 6.9, respectively.

4.2 Transaction Management

eBPF's execution environment limits the possible concurrency control protocols in BPF-DB. Each operator is a self-contained eBPF program and, thus, must contain all concurrency control logic within it. Also, the protocol's implementation must be simple enough to get past the eBPF verifier. For example, OCC would not work because its commit logic that reevaluates database operations against others to detect conflict is non-deterministic. As described in Section 2.2, eBPF's inability to nest spin latches or call eBPF helper functions while holding a latch disqualifies protocols with complex mutual exclusion logic. Lastly, eBPF programs are event-driven and cannot yield their thread or suspend execution. This requirement precludes protocols requiring a centralized scheduler that can block threads on lock requests [58, 66].

BPF-DB uses a decentralized multi-versioned Strict Two-Phase Locking (Strict-2PL) protocol for serializable transactions [23]. Operators acquire read and write locks that reside inline with the records rather than in a centralized lock table [37]. Strict-2PL requires transactions to hold all their locks until they commit or roll back. BPF-DB also employs a multi-version mixed optimization (ROMV) whereby read-only transactions do not acquire any read locks but still maintain serializable properties [23, 69].

BPF-DB's `timestamps` map is the global synchronization point for concurrency control and contains only a single element for operators to access. This object contains two fields: (1) BPF-DB's current global timestamp, and (2) the timestamp of the oldest running transaction in the system. The struct also contains a spin latch used to provide mutual exclusion for the `COMMIT` operator. The `BEGIN` operator uses the `timestamps` map to get the current timestamp at transaction start time, and stores that in the session context for the duration of the transaction. The `COMMIT` operator again uses the `timestamps` map—this time to increment the global current timestamp for the newly

committed transaction, and to store the timestamp in the index for all modified versions. These steps all take place under the `timestamps` map entry's latch to prevent simultaneous committers.

Locking-based concurrency control introduces the potential for deadlocks, and DBMSs typically implement either (1) deadlock detection or (2) deadlock prevention. Deadlock detection generates dependency graphs and then applies a cycle detection algorithm—logic too complex for the eBPF verifier. Instead, BPF-DB employs deadlock prevention by allocating timestamps for multi-versioning but does not use these timestamps to implement a wound-wait or wait-die deadlock policy. These policies require coordination between workers to implement lock stealing and coordinate restarts, and eBPF does not provide primitives for this behavior. Instead, BPF-DB applies a no-wait policy for lock acquisition whereby `SET` operators that encounter a lock conflict immediately report an Error state—regardless of their timestamp's age relative to the transaction holding the lock. This approach provides the fastest transaction performance for in-memory DBMSs [71].

4.3 Write-Ahead Logging

In-memory DBMSs rely on two methods to guarantee that they can restore their database contents after a restart: logging and checkpointing. The challenge, however, is that these components are the least amenable to applying user-bypass. This conflict stems from eBPF's inability to write to disk. I/O operations like writes can block for an indeterminate amount of time, which is not allowed during the execution of eBPF programs.

WAL requires the DBMS to maintain a ledger of all database modifications. Upon restart, the DBMS replays the log to reconstruct the database contents. Most DBMSs expose a knob that determines whether transactions commit synchronously or asynchronously. With asynchronous durability, the DBMS submits a transaction's changes to the log writer but can acknowledge transaction completion to the client without guaranteeing that the changes are persistent on disk. Synchronous durability ensures the OS has written a transaction's contents via the `write()` system call. It may also guarantee that the system performs the `fsync()` system call to ensure the OS flushes its write buffers.

eBPF's inability to initiate I/O (i.e., disk or network writes) necessitates a user-space component to persist database contents. Thus, BPF-DB takes a hybrid user-space and kernel-space approach to logging. If WAL is enabled, BPF-DB sends all write operations to a ring buffer accessible from user-space. If the ring buffer is full and BPF-DB cannot complete its write operation, it reports an error and tail calls the error continuation. The application must roll back the transaction because not all its writes are in the log.

As shown in Figure 2, a WAL consumer process in user-space consumes the contents of the ring buffer. eBPF's ring buffer offers the ability to notify user-space consumers via `epoll()` when to consume the buffer. BPF-DB offers a tunable knob whereby it will only notify waiters after writing a specified amount of data. This approach reduces the amount of signal handling overhead between kernel-space and user-space, and can be adjusted based on the desired wakeup frequency of any user-space consumers.

After wakeup by notification or a timeout, the user-space component consumes data from the buffer, providing asynchronous durability for BPF-DB writes. This durability guarantee follows the default settings of other in-memory DBMSs like SingleStore, as well as embedded DBMSs like SQLite and RocksDB [6, 17, 21]. BPF-DB writes timestamps with logical log records and provides an eBPF map entry for WAL consumers to update the last persisted timestamp. With this approach, client applications implement their durability semantics and can optionally check if their writes are durable before proceeding. The fixed-sized ring buffer and the timestamp for most recently flushed writes present two possible mechanisms for applications propagate back pressure to a workload. We evaluate the trade-offs of this logging approach in Section 6.3.

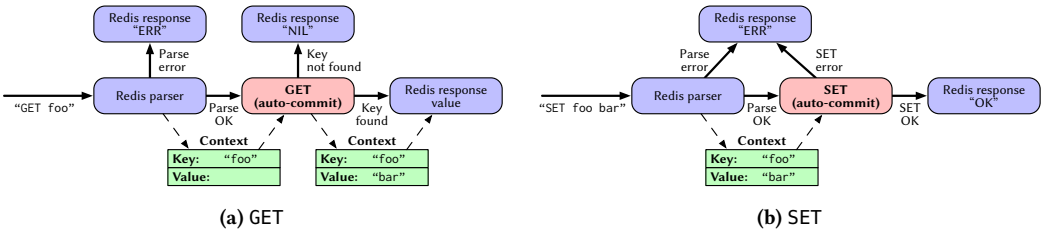


Fig. 5. Networked Key-Value Store CFG – The transactional KVS composes BPF-DB's `GET` and `SET` commands (shown in red) with application logic (shown in blue). Conditional tail calls define the continuations for each eBPF program.

Checkpointing is the process of writing all the contents of the database to disk, thus preserving a consistent snapshot of the database. The most naïve checkpointing approach is to block all transactions that are not read-only while flushing the database contents to disk. Once the checkpoint is complete, all transactions can resume. For BPF-DB, one approach to stop all transactions would be to unattach all eBPF programs so they could no longer be triggered. However, writing all of the database's contents remains nontrivial. Techniques that rely on virtual memory's copy-on-write to create a database snapshot by calling `fork()` on the DBMS's process will not work with a user-bypass system because BPF-DB stores its database in kernel memory [42]. Instead, a user-space process must read the eBPF maps' contents using the `bpf()` system call.

BPF-DB's CPS control flow presents new opportunities to implement different checkpointing techniques. For example, the Linux kernel can atomically swap eBPF programs at runtime. One way to implement blocking checkpointing is to swap out BPF-DB's `SET` operators with eBPF programs that call the error continuation while checkpointing is in progress. Once the checkpoint is complete, the DBMS reverts `SET` operators to their original version, and applications can resume writing to the database. More sophisticated fuzzy checkpointing could build on this technique by temporarily swapping in `SET` operators that write to an alternate set of eBPF maps while a checkpoint is in progress [57]. The DBMS would also need to swap `GET` operators to be able to look in the correct eBPF map for their values. After the checkpoint is complete, the DBMS switches back to the original operators. We defer the exploration of runtime operator modification and checkpointing to future work.

5 BPF-DB Applications

Section 3 describes BPF-DB as an embedded DBMS that enables developers to build eBPF applications around its operators. We now detail two sample applications we use to demonstrate and evaluate BPF-DB. They adhere to the archetype described in Section 2.4: (1) a DBMS that uses BPF-DB as its transactional storage manager, exposed to clients as a *networked key-value store*, and (2) an application that uses BPF-DB to execute *multi-statement transactions*.

5.1 Networked Key-Value Store

The first application we use to evaluate BPF-DB is a networked key-value store (KVS) that supports a modified Redis network protocol. We implement an eBPF program that parses a subset of Redis commands and types with continuations to call into BPF-DB operators. The KVS's parser supports `SET` and `GET` commands. We also extended the protocol with `BEGIN`/`COMMIT` commands for transactions because Redis' `MULTI`/`EXEC` commands do not provide serializable ACID semantics. Redis is a string-based protocol, so the transactional KVS stores all keys and values in their native

Redis encoding. We also include programs that serialize and send Redis protocol responses over the network to the client as continuations.

Figure 5 shows the CFGs for the GET and SET operations. In Figure 5a, the query "GET foo" is input to the first piece of application code, the Redis parser. If the input string parses as a GET command, the application populates BPF-DB's context object with the key and tail calls into BPF-DB's GET operator. GET has two possible outcomes: (1) the key is not found, in which case BPF-DB tail calls to an eBPF program that responds with Redis's NIL object, or (2) the key is found, and the continuation program serializes the value response to the client. Figure 5b shows similar behavior but with continuations that reflect SET's possible continuations.

5.2 Multi-Statement Transactions

The second application exercises BPF-DB's serializable transactions with multiple GET and SET operators per transaction. We implement a stored procedure from the Voter benchmark, which models a call-in voting system [64]. We only run the VOTE stored procedure and drop any views and procedures related to results tallying. The schema consists of (1) CONTESTANTS table that maps contestant identifiers to contestant names, (2) VOTES table that records the phone number, state, and contestant for each vote, and (3) AREA_CODE_STATE that maps phone area codes to their states. We describe the VOTE stored procedure below:

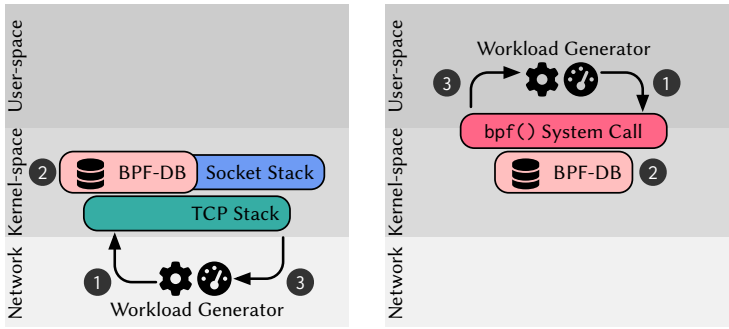
- The workload generator provides three arguments. (1) The ten-digit phone number's area code is selected uniformly, with the seven remaining digits generated from a uniform distribution. (2) The contestant identifier is uniformly chosen between one and twelve, but approximately 1% of VOTE operations contain an invalid contestant to exercise the validation logic. (3) The maximum number of votes per phone number.
- Look up the phone area code to determine its state.
- Look up the contestant identifier to check its validity.
- Look up the phone number to check if it's over the vote limit.
- Update the phone number's vote count and record a new vote.

Because of its KVS interface, BPF-DB does not expose explicit schemas or tables. Instead, BPF-DB's Voter application prepends keys with distinct prefixes so entries in different tables map to distinct key spaces. To mimic the view for the number of votes by each phone number, BPF-DB's implementation defines a separate keyspace and explicitly maintains an entry for each phone number. This approach requires an extra SET for BPF-DB compared to the VOTE procedure's SQL definition.

6 Evaluation

We next evaluate BPF-DB for the two applications that we described in Section 5 compared against state-of-the-art DBMSs. First, we attach a eBPF program (i.e., Redis parser) in the socket layer of the network stack (Figure 6a). When a buffer arrives on specified sockets, the Redis parser and subsequent BPF-DB continuations will run. To eliminate external bottlenecks, we also run workloads where a user-space thread invokes BPF-DB operations (Figure 6b).

We evaluate BPF-DB on servers that contain 2×20-core Intel Xeon Gold 5218R CPUs, 192 GB DRAM, a Samsung PM983 SSD, and a dual-port 10GbE network adapter. All servers run Ubuntu Linux 22.04 (kernel v5.15) and connect to a Cisco Nexus 3064 10GbE network switch. Unless otherwise specified, we use separate servers for DBMSs and workload generators. We pin user-space programs to a single CPU socket to avoid NUMA effects. We configure the NIC to use 20 receive queues, each with affinity set to a dedicated CPU core on the local NUMA node. Lastly, we



(a) Networked Workload **(b)** Local Workload

Fig. 6. Workload Generators – We exercise BPF-DB in two ways: (1) as applications attached at the socket layer of the kernel's network stack as in Figure 6a, driven by an external tools over the network, and (2) loaded into kernel-space not attached to any events as in Figure 6b. Instead, we drive BPF-DB with the `bpf()` system call from user-space threads.

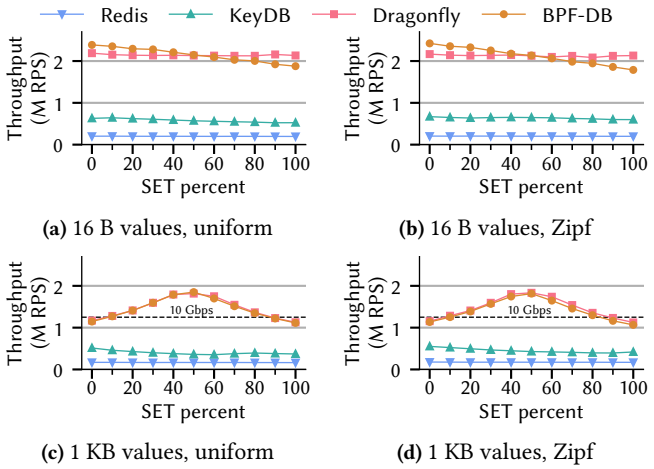
optimize the network stack by disabling interrupt rate limiting, expanding network buffer sizes, and turning off TCP timestamps and selective acknowledgements.

We compare BPF-DB's transactional KVS against three other open-source Redis-compatible proxies:

- **Redis (v7.2):** This is is one of the most widely deployed in-memory key-value stores. We use Redis as the baseline implementation for user-space key-value stores.
- **KeyDB (v6.3):** This is a fork of Redis that adds user-space worker threads and coarse-grained latches to maintain consistency across workers. It allows us to evaluate the Redis architecture with added parallelism. We configured KeyDB to use four worker threads, as their documentation does not recommend increasing beyond this number due to latch contention.
- **Dragonfly (v1.14):** This multi-threaded KVS is a state-of-the-art in-memory implementation of the Redis protocol. Dragonfly employs key-space partitioning, lightweight user-space threads (i.e., fibers), asynchronous I/O, and a locking scheme inspired by VLL to maintain consistency across workers [58]. We configure Dragonfly to use 20 worker threads to match the number of physical CPUs on its NUMA node.
- **VoltDB (v11.4):** This is an ACID-compliant in-memory DBMS that supports serializable transactions via horizontal partitioning (i.e., sharding) and pre-compiled Java stored procedures [15]. We configure VoltDB to use 20 partitions to maximize its performance when pinned to 20 dedicated CPU cores.

For our networked experiments, we use *memtier* for workload generation and modify its key generation to support skewed Zipf distributions ($\theta = 0.99$) [18]. We load 10M keys for each experiment and use 16 B and 1 KB value sizes to reflect caching workloads and YCSB, respectively. memtier submits the workload in a closed loop across 1,000 simultaneous clients on 80 worker threads [30]. Each configuration runs five times. We run these Redis workloads run with logging and checkpointing disabled because each DBMS provides different durability semantics. For this reason, we evaluate BPF-DB's WAL characteristics in isolation in Section 6.3.

In scenarios where network overheads limit evaluating BPF-DB, we run its applications on local threads as shown in Figure 6b. These threads use the `bpf()` system call and its `BPF_PROG_TEST_RUN` argument to run BPF-DB in a controlled fashion. Compared to the networked scenario in Figure 6a, there are no software interrupts for the kernel to schedule, no extra buffers to copy, and no network protocols or NIC driver overheads.



(a) 16 B values, uniform **(b)** 16 B values, Zipf

(c) 1 KB values, uniform **(d)** 1 KB values, Zipf

Fig. 7. KVS Throughput (Networked) – Total requests per second over 10GbE network (Redis protocol) while varying the ratio of `SET` commands from read-only to write-only. Scenarios differ in value size (16 B or 1 KB) and key distribution (uniform or Zipf).

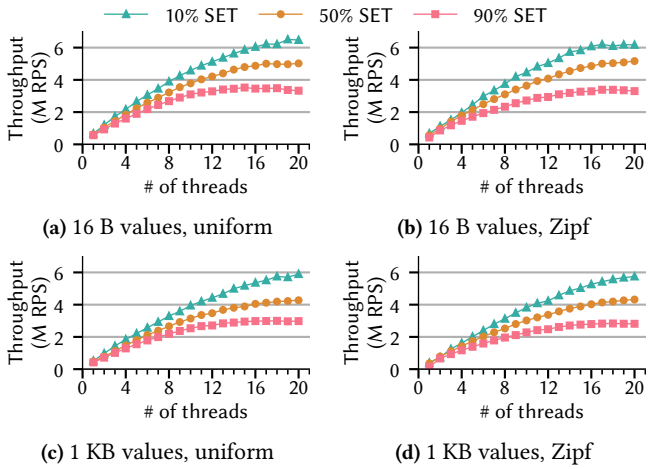
6.1 KVS Throughput (Networked)

We first measure how BPF-DB performs as a transactional KVS over the network for different value sizes and workloads. We compare BPF-DB against other Redis-compatible DBMSs for mixed `GET` and `SET` workloads. We attach the networked KVS described in Section 5.1 at the kernel's socket layer. When a message arrives, the DBMS (1) parses the message for a valid `GET` or `SET` command, (2) parses the key and (optional) value, (3) tail calls into the corresponding BPF-DB operation, and (4) tail calls into continuations that construct and send the response to the network client.

For each value size, we vary the workload's read and write ratios. Increasing the number of writes reduces a DBMS's scalability due to memory allocations, data structure synchronization (e.g., latches), and concurrency control protocols. For this reason, we scale the ratio of `SET` operations from 0% (i.e., read-only) to 100% (i.e., write-only) in 10% increments and measure the maximum throughput.

Figure 7a shows the throughput for 16 B values with uniform random key generation. As a single-threaded user-space application, Redis achieves the lowest performance with 202 K requests per second (RPS) for the read-only scenario, dropping 3% to 196 K RPS as `SET`s increase. KeyDB's thread-based parallelism enables it to outperform Redis, albeit with a speedup that is not linear to its four worker threads. KeyDB starts at 629 K RPS for the read-only scenario, dropping 17% to 524 K RPS in the write-only scenario. Redis and KeyDB exhibit similar trends in the other scenarios (i.e., value size, key distribution) of Figure 7, and due to their non-competitive throughput (<1 M RPS), we omit further discussion of them.

For the read-only scenario in Figure 7a, Dragonfly processes 2.19 M RPS while BPF-DB achieves 2.39 M RPS, a 9% increase in throughput. As writes increase, the performance lead flips: KeyDB yields 2.13 M RPS and BPF-DB performs 1.88 M RPS in the write-only scenario. Dragonfly's user-space data structures are more efficient than BPF-DB's kernel-space eBPF maps, which require internal latches and RCU synchronization. As writes increase, these synchronization methods hurt BPF-DB's throughput. Figure 7b exhibits similar trends, demonstrating the performance of BPF-DB's strict-2PL concurrency control and bounded version vectors, even under high contention scenarios like skewed key distributions.



(a) 16 B values, uniform **(b)** 16 B values, Zipf

(c) 1 KB values, uniform **(d)** 1 KB values, Zipf

Fig. 8. KVS CPU Scalability (Local) – Total requests per second running BPF-DB locally while varying the number of CPU cores dedicated to BPF-DB. We measure three ratios of `SET` commands, and scenarios differ in value size (16 B or 1 KB) and key distribution (uniform or Zipf).

In Figures 7c and 7d, Dragonfly and BPF-DB with 1 KB values exhibit different behavior compared to 16 B values. These results demonstrate that Dragonfly and BPF-DB saturate the network bandwidth limits of the test environment, as shown by the annotation denoting 10 Gbps. At read-only and write-only extremes, the network limits DBMS throughput to ~10 Gbps. More diverse workloads benefit from bidirectional bandwidth, allowing the DBMSs to saturate both inbound and outbound network traffic. Throughput peaks when `GET` and `SET` each make up half of the requests,though performance is not able to reach the theoretical limit of 20 Gbps due to TCP/IP protocol overhead. As a backing store for a networked KVS, BPF-DB remains competitive with Dragonfly and offers fully serializable transactions.

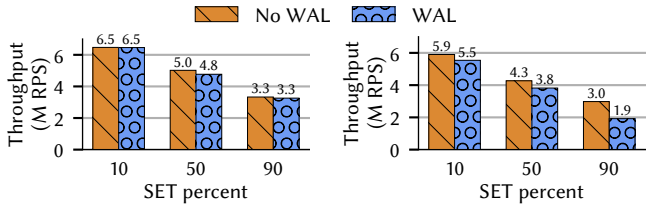
6.2 KVS CPU Scalability (Local)

We next remove the network bottleneck from Section 6.1 to find BPF-DB's maximum throughput. We use the `bpf()` system call and its `BPF_PROG_TEST_RUN` argument to trigger BPF-DB `GET` and `SET` operators from user-space threads. We scale the number of CPU threads running in user-space to understand BPF-DB's scalability under increasing resource (e.g., CPU, memory, lock) contention. We load 10 M keys before test execution and run the workload with different ratios of `SET` operations, value sizes, and key distributions. We will use this configuration for the remaining KVS experiments.

Figure 8 shows the BPF-DB's performance when driven by local CPU threads. The 90% `SET` workload plateaus earlier than the less write-intensive workloads, demonstrating the overhead of eBPF maps' RCU synchronization. In the 16 B values and uniform key scenario, 10% `SET` levels off at 6.5 M RPS, continuing to scale almost to the number of CPU cores (20). 90% `SET` reaches maximum performance near 3.5 M RPS much earlier, where synchronization overheads limit CPU core utilization to 15 of the total 20 cores.

Changing the key distribution from uniform to Zipf does not introduce a significant performance drop. The added lock contention and GC work are not the bottleneck; instead, they highlight again that the eBPF maps limit performance.

Increasing the value size to 1 KB reduces throughput due to the increased time spent copying values to and from BPF-DB. In the 1 KB values and uniform key distribution scenario, 10% `SET` plateaus at 5.9 M RPS, 9% lower than the test with smaller value sizes. 90% `SET` levels off at 3 M RPS,



(a) 16 B values, uniform **(b)** 1 KB values, uniform

Fig. 9. Write-Ahead Logging – Total requests per second running BPF-DB locally with WAL disabled and enabled. We measure three ratios of SET commands, and scenarios differ in value size (16 B or 1 KB) with a uniform key distribution to reduce contention and maximize throughput.

this time a 14% drop compared to smaller value sizes. Larger value sizes introduce longer critical sections in the eBPF maps; thus, the write-intensive workload suffers from a larger performance drop. This analysis mirrors Linux's warnings that RCU is not a scalable synchronization technique when writes constitute more than 10% of the workload [19]. However, eBPF does not expose an alternative synchronization method for maps, so BPF-DB must use the provided interface.

6.3 Write-Ahead Logging Throughput (Local)

We now measure the performance impact of BPF-DB's WAL architecture (see Section 4.3). To maximize BPF-DB's requests per second to exercise its WAL code path and reduce the impact of other DBMS component, we only generate keys from a uniform distribution. Zipf key distribution accesses a small number of keys, generating lock contention and GC overhead.

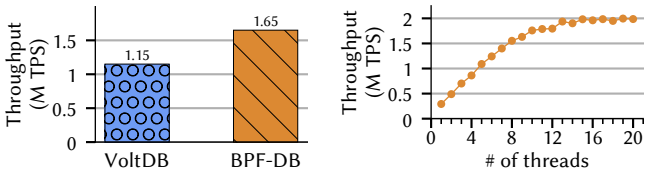
We run BPF-DB's local KVS with 20 worker threads with multiple SET ratios with and without logging enabled. The user-space log consumer immediately removes data from the shared ring buffer and does not write log records to disk to avoid interference. We configured BPF-DB to use a 512 MB ring buffer and have the DBMS notify any processes waiting on the ring buffer's file descriptor after writing 2 MB. The user-space consumer has a timeout of 1 ms if it does not receive a notification to consume the buffer.

Figure 9 shows BPF-DB's throughput with WAL enabled and disabled. The 90% SET scenario with both 16 B and 1 KB values induces the most writes and, thus, is the most taxing on the WAL architecture. With 16 B values, the write-heavy scenario drops from 3.33 M RPS to 3.26 M RPS, showing that WAL incurs a 2% overhead. None of those operations incur a roll-back due to a full ring buffer, so we attribute this overhead exclusively to the additional instructions for copying and submitting write operations into the ring buffer.

The 90% SET 1 KB scenario suffers a 35% drop in performance with WAL enabled, going from 2.98 M RPS to 1.92 M RPS. The larger value sizes require BPF-DB to spend more CPU cycles copying data into the ring buffer. This workload averages ~1.4 M roll-backs per second due to a full ring buffer. This experiment demonstrates the limitations of BPF-DB's user-space WAL design, where the user-space consumer cannot exceed ~2 GB/s. We could enhance BPF-DB for write-heavy scenarios using parallelism optimizations similar to Silo-R: multiple ring buffers could have multiple consumers if the host can write more than 2 GB/s to storage [67]. We defer this investigation for future work.

6.4 Muti-Statement Throughput (Networked)

We next evaluate BPF-DB's multi-statement transaction performance compared to VoltDB using the Voter benchmark. For VoltDB, we use its asynchronous workload generator to maximize throughput over the network. We modified the generator to use multiple threads because one



(a) Throughput (Networked) **(b)** CPU Scalability (Local)

Fig. 10. Multi-Statement Throughput and Scalability – Networked scenario measures transactions per second over 10GbE network compared to VoltDB. Local scenario measures transactions per second running BPF-DB locally while varying the number of CPU cores dedicated to BPF-DB.

instance cannot saturate our VoltDB server. For BPF-DB, we embed a stored procedure that contains `BEGIN`, three `GET`s, two `SET`s, and `COMMIT`/`ROLLBACK` operations stitched together through tail calls. As in Section 6.1, we attach the stored procedure in the kernel's socket layer and prepend BPF-DB's operations with a parser to extract the arguments. We modify memtier invoke the stored procedure with a custom Redis command.

Figure 10a shows that BPF-DB processes 1.65 M Voter transactions per second (TPS) and achieves 43% higher throughput than VoltDB. Contrast this with BPF-DB's ~2 M RPS KVS throughput in Figure 7. We attribute this difference to the cost of the extra eBPF map operations to read and update multiple values. Despite its partitioned and lock-free architecture, when we profile VoltDB we find that it spends significant time in the JVM's futex and synchronization calls and user-space thread scheduling.

6.5 Muti-Statement Scalability (Local)

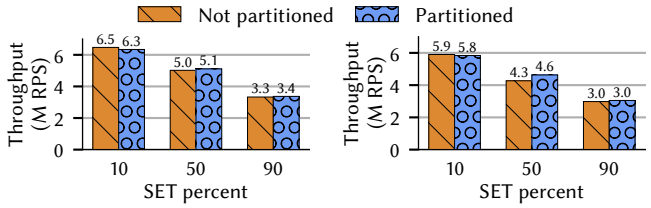
Like in Section 6.2, for our next experiment we remove network overheads by driving the Voter workload with user-space threads using the `bpf()` system call. We measure BPF-DB's maximum throughput for serializable transactions with multiple statements. We also show BPF-DB's scaling characteristics by sweeping the number of user-space threads that invoke the stored procedure.

Figure 10b shows BPF-DB's maximum TPS for the Voter workload while increasing the number of threads. Like the 90% `SET` workload in Figure 8, Voter is write-intensive and plateaus when using around 15 of the 20 available CPU cores. However, the maximum throughput is lower than that experiment due to the extra BPF-DB operations, eBPF map lookups, and commit logic. We find that BPF-DB achieves ~2 M TPS on a single 20-core CPU.

To better understand BPF-DB's behavior in this workload, we use BCC's `profile` tool to sample BPF-DB's call stacks [5]. Our analysis indicates that BPF-DB spends most of its CPU time executing code not in its programs, but rather in kernel-provided eBPF helper functions. For example, 78.5% of the stack trace samples are in eBPF map operations (e.g., lookup, update), 4.6% are in the kernel's hash function (i.e., `jhash`), 4.8% are in the kernel's spin-lock implementation, and 9.5% are in helpers that copy data between kernel-owned memory. We leave optimizing BPF-DB further to reduce its reliance on kernel-provided helpers as future work.

6.6 Checkpointing

We next evaluate the performance of persisting BPF-DB's database contents to disk. As discussed in Section 4.3, eBPF programs cannot initiate their own disk access, so we rely on user-space processes to write eBPF maps to files. In this scenario, we first run the same Voter workload as in Section 6.4 for ~50 M transactions. Then, we simulate a blocking checkpoint by quiescing the system. We use `bpftool`, which can view and modify eBPF maps from user-space, to dump BPF-DB's database



(a) 16 B values, uniform **(b)** 1 KB values, uniform

Fig. 11. Partitioned Storage – Total requests per second running BPF-DB locally with and without key space partitioning. We measure three ratios of `SET` commands, and scenarios differ in value size (16 B or 1 KB) with a uniform key distribution to reduce contention and maximize throughput.

contents to files [20]. We time the duration for each eBPF map and report the sum. The average of five iterations of this operation was 1,068 seconds.

This value serves as an upper-bound for checkpointing BPF-DB. First, `bpftool`'s dump method performs inefficient map access due to Linux's limited eBPF user-space API. Reading an entire eBPF map requires iterating through each key-value pair with a distinct system call. Array-based eBPF maps support access from user-space via the `mmap()` system call which could improve read performance, but BPF-DB primarily relies on maps backed by hash tables that do not support `mmap()`. Second, this approach incurs significant write amplification. We do not perform a system-wide GC process before checkpointing, so this process can write obsolete and redundant versions to disk. Furthermore, index entries contain unnecessary metadata like version arrays with timestamp values that would not be necessary if old versions were preemptively garbage collected. Lastly, `bpftool` dumps maps in a human-readable JSON format which results in files that are larger than necessary to recover database contents. We defer a more sophisticated checkpointing scheme for future work with this approach as a baseline.

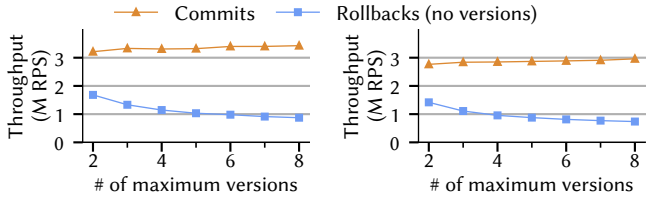
6.7 Partitioned Storage (Local)

To reduce the kernel's synchronization overheads in BPF-DB, we horizontally partition the database by replacing the system's top-level index with four indexes that each have their own value storage eBPF maps. At runtime, the DBMS uses a fast hash function to determine the partition for a given key. We use the XXH64 function because it has good performance and passes the eBPF verifier [11]. Faster hash functions either are too complex to satisfy the verifier or rely on SIMD instructions (e.g., AVX2) that eBPF does not support. We run BPF-DB locally using the KVS workload and generate keys from a uniform distribution to maximize throughput.

Figure 11 shows BPF-DB's throughout with varying `SET` ratios. The DBMS's performance is lower in the read-heavy workload (i.e.,10% `SET`) with 16 B and 1 KB values. Once `SET` commands exceed 50%, partitioning yields a slight throughput benefit, but the increase never exceeds 8%. BPF-DB's performance does not significantly improve with partitioning because it does not spend most of its time executing code that benefits from the reduced map contention.

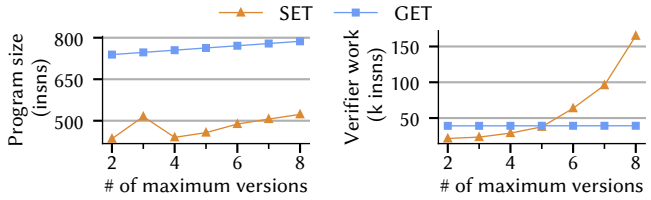
6.8 Scaling MVCC Versions (Local)

Due to the verifier's constraints, BPF-DB supports MVCC with a fixed number of versions per database entry. If one of BPF-DB's `SET` operations attempts to write to a value that has reached its maximum number of versions, it must attempt cooperative GC or roll back (see Section 4.1). In this experiment, we measure the impact of this compile-time constant while running a write-heavy workload (90% `SET`) using a skewed key distribution (i.e., Zipf).



(a) 16 B values, Zipf **(b)** 1 KB values, Zipf

Fig. 12. Scaling Maximum Versions – Total requests per second running BPF-DB locally while varying the number of versions per database entry. Scenarios differ in value size (16 B or 1 KB), and we maximize version generation by using the 90% SET workload with a skewed key distribution.



(a) eBPF program size **(b)** Verifier instructions

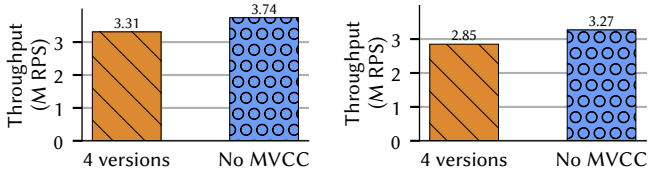
Fig. 13. Verifier Overhead for Versioning – The program size and verifier work for BPF-DB's `GET` and `SET` operators while varying the maximum number of versions per database entry.

Figure 12 shows BPF-DB's performance when changing the number of versions per entry. For 16 B values, BPF-DB with two versions supports 3.2 M RPS and 1.7 M rollbacks per second from no available versions, while eight versions per key yields 3.4 M RPS and 0.9 M rollbacks per second. Although this change halves the number of rollbacks per second, there is only a 6.6% increase in overall throughput. At maximum throughput of over 3 M RPS on a small number of keys, increasing the number of keys does little to change how quickly BPF-DB exhausts its available versions. The results for 1 KB values are similar, demonstrating that the DBMS is not significantly affected by the number of versions.

In addition to altering BPF-DB's runtime behavior, changing the number of versions also affects both program size and the amount of work that the eBPF verifier expends. Recall that we limit the maximum number of versions because the verifier requires compile-time constants for loops, and explores all iterations and branches during verification. We use `bpftool` to get eBPF program size and the number of instructions processed by the verifier [20].

We see in Figure 13a that the number of eBPF instructions for the `GET` operator scales linearly with size of the version array. This behavior is due to the loop that compares timestamps in the entire array to find the correct value. The compiler unrolls the loop to improve performance, thus increasing instruction count. In contrast, `SET`'s number of instructions oscillates before scaling linearly. When the compiler generates eBPF code for `SET` with three versions, it optimizes the C code differently and results in increased instruction count. This oscillation shows the difficulty in anticipating how compilers like Clang and GCC generate optimized eBPF bytecode.

The `GET` operator contains more instructions than `SET` despite its less complex logic (e.g., it does not perform GC). This higher instruction count is because it copies data out of the value map. `SET` does not require these instructions because the logic to copy a value into a map is handled by the eBPF helper that inserts the value into the map. Looking up a value in a eBPF map only returns



(a) 16 B values, Zipf **(b)** 1 KB values, Zipf

Fig. 14. Single Version Storage – Total requests per second running BPF-DB locally with MVCC and without. Scenarios differ in value size (16 B or 1 KB), and we maximize version generation by using the 90% SET workload with skewed (i.e., Zipf) key distribution.

a pointer to the value, which GET in turn copies at its instruction cost into the BPF-DB Context object.

Figure 13b shows the number of instructions that the verifier processes when loading GET and SET operators with different numbers of versions. GET scales linearly because the number of states that the verifier inspects increases as the limit on the loop that scans the version array increases. However, SET's verifier complexity increases exponentially as the maximum number of versions increases. SET contains more complex logic inside of the loops that scan the version array (e.g., cooperative GC). Though not shown in the figure, increasing the number of versions past eight causes the verifier to reject the SET program. These results demonstrate the non-obvious lack of correlation between eBPF program size and the amount of work the verifier performs to load a program.

6.9 Single Version Storage (Local)

Lastly, we evaluate a version of BPF-DB without versioning. For single-statement transactions, maintaining versions is unnecessary because they do not need a consistent snapshot of multiple entries. Removing MVCC allows BPF-DB to elide version chain traversal, timestamp allocation, and GC. Because Strict-2PL requires read-only transactions to acquire locks, we augment BPF-DB's lock acquisition logic to retry up to 4096 times, which is within the bounds of the verifier. We evaluate this design with the high-contention workload from Section 6.8: 90% SET with Zipf key distribution.

Figure 14 shows BPF-DB's throughput with and without MVCC. The reduced complexity of the SET and GET operators enables higher overall throughput, despite GET now requiring a lock acquisition. This single-statement workload shows notable speedup from this change, increasing throughput by 13% and 15% for 16 B values and 1 KB values, respectively. But multi-statement transactions with conflicting access sets would incur more roll-backs. Developers may decide to make this trade-off if transactions are unlikely to conflict because their access sets are either small or unlikely to overlap.

We conclude our analysis of MVCC versions by combining the approaches of Sections 6.8 and 6.9 to measure the impact on GET and SET operator latency. We fix the number of requests at 100 K RPS and measure operator latency in a high contention scenario that generates many versions (90% SET, Zipf keys). Table 2 show the results of measuring operator latency while changing the maximum number of versions. Removing MVCC entirely (i.e., one maximum version) increases p25, median, and p75 latency in exchange for a lower p99 latency. This scenario removes all BPF-DB logic related to GC as well, which contributes to the large tail latency reduction.

Table 2. Maximum Versions and Latency (1 KB) – Average operator latency (in microseconds) running BPF-DB locally with MVCC and without. This scenario uses 1 KB values, and we maximize version generation by using the 90% SET workload with skewed (i.e., Zipf) key distribution.

Max versions	Min	p25	p50	p75	p99
1 (No MVCC)	0.68	1.46	1.7	2.0	5.32
2	0.65	1.32	1.55	1.83	46.21
3	0.53	1.25	1.51	1.86	45.04
4	0.52	1.22	1.49	1.87	44.03
5	0.52	1.18	1.44	1.83	39.56
6	0.52	1.16	1.41	1.78	40.89
7	0.52	1.14	1.39	1.75	37.32
8	0.52	1.13	1.38	1.72	37.25

7 Related Work

OS extensibility has been a topic for researchers for decades [24, 59, 60]. More recently, the ExtOS Linux prototype reduces data movement in the kernel's stack by extending the `read()` system call to support database operations via a kernel module [22].

Unikernels is an approach where applications are compiled with a library OS to yield an application-specific machine [32, 41, 43, 68]. Unikernels remain an active and promising research area, but the industry has been slow to adopt this approach. Recent work argues that cloud virtualization and their hypervisor abstractions provide new opportunities for DBMSs designed as unikernels [45].

LIBDBOS tackles for a recurring problem: enabling user-space DBMSs to manage system resources more efficiently than the OS, this time by presenting new kernel-bypass abstractions to increase user-space DBMS performance [75]. Their technique applies novel virtual memory techniques to provide high-performance snapshots for HTAP workloads and a kernel-resident buffer manager. The evaluation demonstrates how a hardware and OS-aware design for kernel-bypass can reduce the overhead of TLB-shootdowns and improve overall DBMS performance.

The XRP project applies user-bypass to the Linux kernel's storage stack [74]. For example, B+tree lookups read multiple disk pages before finding the destination leaf node. XRP pushes DBMS logic into the NVMe driver via eBPF. With XRP, the DBMS performs B+ tree node traversal in kernel-space by resubmitting multiple NVMe operations. XRP's use of user-bypass reduced the amount of data copied to user-space and the number of repeated system calls, thereby demonstrating the latency and throughput benefits it achieves versus kernel-bypass using DPDK and asynchronous I/O.

BPF-oF is a storage protocol built on top of NVMe-oF that allows applications to send predicates as custom eBPF functions to a remote storage server to filter data before transferring [72]. This approach builds upon XRP to enable application logic to run in the NVMe layer of disaggregated storage nodes, and addresses cache coherency between distributed nodes [74].

BMC is a a Memcached-compatible in-memory key-value store that uses eBPF to accelerate `GET` operations over UDP [34]. BMC presents a eBPF map-based cache below the kernel networking stack. During a `GET` operation, the eBPF program first checks the map. If there is a cache miss, the request passes to Memcached which serves the request and then updates the cache. BMC can serve subsequent requests for the same key from the cache without accessing user-space.

TScout uses runtime code-generation and DBMS source code annotation to generate eBPF programs for online training data collection for self-driving DBMSs [27]. This approach reduces the overhead of collecting system resource metrics with repeated system calls. TScout keeps the

training data in kernel-space during query execution, and later collects the results off the critical path.

Tigger is a DBMS proxy that applies user-bypass to accelerate operations like connection pooling and workload mirroring [28]. Tigger's eBPF programs parse the PostgreSQL protocol to safely multiplex transactions across shared connections.

DINT is a distributed, key-value transactional DBMS that runs in kernel-space via eBPF [76]. DINT uses a hybrid approach where common operations run in kernel-space while less common operations run in user-space. The DBMS stores small value sizes in fixed-sized eBPF maps, while larger value sizes that require `malloc()` reside in user-space. DINT also uses a hybrid logging and recovery implementation with user-space components for log replay.

McObject sells eXtremeDB as a user-space embedded DBMS that also offers a kernel mode [36]. eXtremeDB's user-space library communicates with a kernel module containing a proxy, the DBMS, and runtimes for network and storage drivers. As discussed in Section 2, unsigned kernel modules require users to disable Secure Boot—a mitigation technique against firmware-based attacks.

OSDB is a new project to embed a relational DBMS in the OS, powered by SQLite [61]. It describes two prototypes: one based on FreeBSD and a subsequent version using Linux. In contrast to BPF-DB, both of OSDB's implementations rely on kernel modules to modify OS behavior. The evalation highlights how a modified version of the `ps()` system call is more performant when supported by a kernel-embedded DBMS, and proposes an iterative approach to extending OS behavior on this new design.

8 Future Work

We demonstrated a design for an embedded DBMS that runs in kernel-space to enable rich user-bypass applications. We believe there are more opportunities to continue this work in varying directions.

Code Generation: Currently, applications developers wanting to use BPF-DB must write their code in a low-level language (e.g., C, Rust), and require significant eBPF knowledge. However, high-level scripting languages that compile down to eBPF (e.g., bpftrace) lower the barrier to entry [9]. Prior work demonstrated the ability to code-generate eBPF programs from templates, which were then specialized to DBMS source code [27]. A domain-specific language (DSL) for eBPF programs that includes BPF-DB semantics would make it easier for developers to write stored procedures like VOTE shown in Section 5. Generating these stored procedures would also enable BPF-DB specialization at code generation time. For example, BPF-DB could elide eBPF maps for value sizes that would not be used or tailor the number of MVCC versions.

Kernel Patches: While this work demonstrates the feasbility and benefits of an embedded DBMS as eBPF programs, it also explores the limitations imposed by the verifier. If BPF-DB were implemented as a new eBPF map type in the Linux kernel with corresponding eBPF helper functions, then some of these limitations would be relaxed. For example, the ability to hold multiple latches would enable metadata (e.g., locks, versions, timestamps) that are currently packed into fixed-size eBPF map values to be located in different memory locations while still being updated atomically. Similarly, transactions could call the kernel's RCU semantics directly which would reduce the current synchronization overheads.

Efficient Checkpointing: Section 6.6 shows BPF-DB's blocking checkpointing performance with a quiesced system. However, this design reads and persists database entries one value at a time, which is inefficient for the user-space application issuing disk writes. One alternative is to use the `mmap()` system call to access eBPF maps' memory, though this approach only works for arrays and

prohibits spin latches. eBPF is evolving; one promising new feature is support for arenas [3]. This new map type allows eBPF programs to build custom data structures (e.g., trees) out of individually managed memory pages. BPF-DB's storage solution could adopt this design, allowing the user-space checkpointing application to access database contents with fewer system calls.

9 Conclusion

We presented BPF-DB, an embedded DBMS that offers serializable, ACID-compliant transactions in kernel-space for user-bypass applications. BPF-DB decomposes its operators into individual eBPF programs that developers can use to build custom procedures. These procedures interleave custom application logic with BPF-DB operators, held together in sequences through tail calls. We demonstrated applications built on BPF-DB and compared them against state-of-the-art counterparts. Our results showed that embedding a DBMS in kernel-space is not only feasible but scalable. Future eBPF applications can use BPF-DB for robust, safe, and high-performance DBMS semantics.

Acknowledgments

This work was supported (in part) by the National Science Foundation (SPX-1822933), VMware Research Grants for Databases, and Google DAPA Research Grants.

References

[1] 2018. Open-sourcing a 10x reduction in Apache Cassandra tail latency. https://instagram-engineering.com/open-sourcing-a-10x-reduction-in-apache-cassandra-tail-latency-d64f86b43589.
[2] 2021. Add bpf_loop helper. https://lwn.net/Articles/877170/.
[3] 2024. [PATCH v2 bpf-next 00/20] bpf: Introduce BPF arena. https://lore.kernel.org/bpf/ZdRNVWhX-7Uel7Gy@infradead.org/T/.
[4] 2024. Use eBPF-based sensor for Microsoft Defender for Endpoint on Linux - Microsoft Defender for Endpoint. https://learn.microsoft.com/en-us/defender-endpoint/linux-support-ebpf.
[5] [n.d.]. bcc/tools/profile.py at master - iovisor/bcc - GitHub. https://github.com/iovisor/bcc/blob/master/tools/profile.py.
[6] [n.d.]. CREATE DATABASE · SingleStore Documentation. https://docs.singlestore.com/db/v8.7/reference/sql-reference/data-definition-language-ddl/create-database/.
[7] [n.d.]. eBPF Instruction Set - The Linux Kernel documentation. https://www.kernel.org/doc/html/v5.17/bpf/instruction-set.html.
[8] [n.d.]. Extend the kernel with eBPF | Android Open Source Project. https://source.android.com/docs/core/architecture/kernel/bpf.
[9] [n.d.]. GitHub - bpftrace/bpftrace: High-level tracing language for Linux eBPF. https://github.com/bpftrace/bpftrace.
[10] [n.d.]. GitHub - cilium/cilium: eBPF-based Networking, Security, and Observability. https://github.com/cilium/cilium.
[11] [n.d.]. GitHub - Cyan4973/xxHash: Extremely fast non-cryptographic hash algorithm. https://github.com/Cyan4973/xxHash.
[12] [n.d.]. GitHub - facebookincubator/katran: A high performance layer 4 load balancer. https://github.com/facebookincubator/katran.
[13] [n.d.]. GitHub - iovisor/bcc: BCC - Tools for BPF-based Linux IO analysis, networking, monitoring, and more. https://github.com/iovisor/bcc.
[14] [n.d.]. GitHub - microsoft/ebpf-for-windows: eBPF implementation that runs on top of Windows. https://github.com/microsoft/ebpf-for-windows.
[15] [n.d.]. GitHub - VoltDB/voltdb: Volt Active Data. https://github.com/VoltDB/voltdb.
[16] [n.d.]. katran/build/fbcode_builder/manifests at main · facebookincubator/katran. https://github.com/facebookincubator/katran/tree/main/build/fbcode_builder/manifests.
[17] [n.d.]. Pragma statements supported by SQLite. https://www.sqlite.org/pragma.html#pragma_synchronous.
[18] [n.d.]. RedisLabs/memtier_benchmark: NoSQL Redis and Memcache traffic generation and benchmarking tool. https://github.com/RedisLabs/memtier_benchmark.
[19] [n.d.]. Review Checklist for RCU Patches — The Linux Kernel documentation. https://www.kernel.org/doc/html/latest/RCU/checklist.html.
[20] [n.d.]. Ubuntu Manpage: bpftool-prog - tool for inspection and simple manipulation of eBPF progs. https://manpages.ubuntu.com/manpages/focal/en/man8/bpftool-prog.8.html.

[21] [n.d.]. WAL Performance. https://github.com/facebook/rocksdb/wiki/WAL-Performance#sync-mode.
[22] Antonio Barbalace, Javier Picorel, and Pramod Bhatotia. 2019. ExtOS: Data-centric Extensible OS. In *APSys*. ACM, 31–39.
[23] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Chapter 5, 160–161.
[24] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan J. Eggers. 1995. Extensibility, Safety and Performance in the SPIN Operating System. In *SOSP*. ACM, 267–284.
[25] Jeff Bonwick. 1994. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *USENIX Summer*. USENIX Association, 87–98.
[26] Jeff Bonwick and Jonathan Adams. 2001. Magazines and Vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources. In *USENIX Annual Technical Conference, General Track*. USENIX, 15–33.
[27] Matthew Butrovich, Wan Shen Lim, Lin Ma, John Rollinson, William Zhang, Yu Xia, and Andrew Pavlo. 2022. Tastes Great! Less Filling! High Performance and Accurate Training Data Collection for Self-Driving Database Management Systems. In *SIGMOD Conference*. ACM, 617–630.
[28] Matthew Butrovich, Karthik Ramanathan, John Rollinson, Wan Shen Lim, William Zhang, Justine Sherry, and Andrew Pavlo. 2023. Tigger: A Database Proxy That Bounces With User-Bypass. *Proc. VLDB Endow.* 16, 11 (2023), 3335–3348.
[29] Paul Chaignon. 2021. The Cost of BPF Tail Calls. https://pchaigno.github.io/ebpf/2021/03/22/cost-bpf-tail-calls.html.
[30] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *SoCC*. ACM, 143–154.
[31] Siying Dong, Shiva Shankar P., Satadru Pan, Anand Ananthabhotla, Dhanabal Ekambaram, Abhinav Sharma, Shobhit Dayal, Nishant Vinaybhai Parikh, Yanqin Jin, Albert Kim, Sushil Patil, Jay Zhuang, Sam Dunster, Akanksha Mahajan, Anirudh Chelluri, Chaitanya Datye, Lucas Vasconcelos Santana, Nitin Garg, and Omkar Gawde. 2023. Disaggregating RocksDB: A Production Experience. *Proc. ACM Manag. Data* 1, 2 (2023), 192:1–192:24.
[32] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole Jr. 1995. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *SOSP*. ACM, 251–266.
[33] Kevin P. Gaffney, Martin Prammer, Laurence C. Brasfield, D. Richard Hipp, Dan R. Kennedy, and Jignesh M. Patel. 2022. SQLite: Past, Present, and Future. *Proc. VLDB Endow.* 15, 12 (2022), 3535–3547.
[34] 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 *NSDI*. USENIX Association, 487–501.
[35] Aakash Goel, Bhuwan Chopra, Ciprian Gerea, Dhruv Mátáni, Josh Metzler, Fahim Ul Haq, and Janet L. Wiener. 2014. Fast database restarts at facebook. In *SIGMOD Conference*. ACM, 541–549.
[36] Andrei Gorine and Alexander Krivolapov. 2020. *A Kernel Mode Database System for High Performance Applications*. Technical Report. McObject.
[37] Vibby Gottemukkala and Tobin J. Lehman. 1992. Locking and Latching in a Memory-Resident Database System. In *VLDB*. Morgan Kaufmann, 533–544.
[38] Brendan Gregg. 2019. *BPF Performance Tools: Linux System and Application Observability* (1st ed.). Addison-Wesley Professional.
[39] Dirk Grunwald and Benjamin G. Zorn. 1993. CustoMalloc: Efficient Synthesized Memory Allocators. *Softw. Pract. Exp.* 23, 8 (1993), 851–869.
[40] Roman Gushchin. 2020. [PATCH bpf-next v9 00/34] bpf: switch to memcg-based memory accounting - Roman Gushchin — fb.com. https://lore.kernel.org/bpf/20201201215900.3569844-1-guro@fb.com/.
[41] Takayuki Imada. 2018. MirageOS Unikernel with Network Acceleration for IoT Cloud Environments. In *ICCBDC*. ACM, 1–5.
[42] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*. IEEE Computer Society, 195–206.
[43] Simon Kuenzer, Vlad-Andrei Badoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Stefan Teodorescu, Costi Raducanu, Cristian Banu, Laurent Mathy, Razvan Deaconescu, Costin Raiciu, and Felipe Huici. 2021. Unikraft: fast, specialized unikernels the easy way. In *EuroSys*. ACM, 376–394.
[44] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. 2011. High-performance concurrency control mechanisms for main-memory databases. *Proc. VLDB Endow.* 5, 4 (dec 2011), 298–309. doi:10.14778/2095686.2095689
[45] Viktor Leis and Christian Dietrich. 2024. Cloud-Native Database Systems and Unikernels: Reimagining OS Abstractions for Modern Hardware. *Proc. VLDB Endow.* 17 (2024).
[46] Justin J. Levandoski, David B. Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang. 2015. Multi-Version Range Concurrency Control in Deuteronomy. *Proc. VLDB Endow.* 8, 13 (2015), 2146–2157.

[47] Alan Maguire. 2023. Introducing bpftune for lightweight, always-on auto-tuning of system behaviour. https://blogs.oracle.com/linux/post/introducing-bpftune.

[48] Yoshinori Matsunobu, Siying Dong, and Herman Lee. 2020. MyRocks: LSM-Tree Database Storage Engine Serving Facebook's Social Graph. *Proc. VLDB Endow.* 13, 12 (2020), 3217–3230.

[49] Steven McCanne and Van Jacobson. 1993. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX Winter 1993 Conference (USENIX Winter 1993 Conference)*. San Diego, CA.

[50] Paul E. McKenney, Joel Fernandes, Silas Boyd-Wickizer, and Jonathan Walpole. 2020. RCU Usage In the Linux Kernel: Eighteen Years Later. *ACM SIGOPS Oper. Syst. Rev.* 54, 1 (2020), 47–63.

[51] Maciej Mościcki and Piotr Rżysko. 2024. Unlocking Kafka's Potential: Tackling Tail Latency with eBPF. https://blog.allegro.tech/2024/03/kafka-performance-analysis.html.

[52] Lily Hay Newman, Matt Burgess, and Andy Greenberg. 2024. How One Bad CrowdStrike Update Crashed the World's Computers. https://www.wired.com/story/crowdstrike-outage-update-windows/.

[53] Michael A. Olson, Keith Bostic, and Margo I. Seltzer. 1999. Berkeley DB. In *USENIX Annual Technical Conference, FREENIX Track*. USENIX, 183–191.

[54] Sujin Park, Irina Calciu, Taesoo Kim, and Sanidhya Kashyap. 2021. Contextual concurrency control. In *HotOS*. ACM, 167–174.

[55] Sujin Park, Diyu Zhou, Yuchen Qian, Irina Calciu, Taesoo Kim, and Sanidhya Kashyap. 2022. Application-Informed Kernel Synchronization Primitives. In *OSDI*. USENIX Association, 667–682.

[56] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *SIGMOD Conference*. ACM, 1981–1984.

[57] Kun Ren, Thaddeus Diamond, Daniel J. Abadi, and Alexander Thomson. 2016. Low-Overhead Asynchronous Checkpointing in Main-Memory Database Systems. In *SIGMOD Conference*. ACM, 1539–1551.

[58] Kun Ren, Alexander Thomson, and Daniel J. Abadi. 2015. VLL: a lock manager redesign for main memory database systems. *VLDB J.* 24, 5 (2015), 681–705.

[59] Felix Martin Schuhknecht, Jens Dittrich, and Ankur Sharma. 2016. RUMA has it: Rewired User-space Memory Access is Possible! *Proc. VLDB Endow.* 9, 10 (2016), 768–779.

[60] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. 1996. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. In *OSDI*. ACM, 213–227.

[61] Robert Soulé, George Neville-Neil, Stelios Kasouridis, Alex Yuan, Avi Silberschatz, and Peter Alvaro. 2025. OSDB: Exposing the Operating System's Inner Database. In *CIDR*. www.cidrdb.org.

[62] Alexei Starovoitov. 2013. LKML: Alexei Starovoitov [PATCH net-next] extended BPF. https://lkml.org/lkml/2013/9/30/627.

[63] Alexei Starovoitov. 2019. BPF at Facebook. https://kernel-recipes.org/en/2019/talks/bpf-at-facebook/.

[64] Michael Stonebraker and Ariel Weisberg. 2013. The VoltDB Main Memory DBMS. *IEEE Data Eng. Bull.* 36, 2 (2013), 21–27.

[65] Gerald J. Sussman and Guy L. Steele Jr. 1998. Scheme: A Interpreter for Extended Lambda Calculus. *High. Order Symb. Comput.* 11, 4 (1998), 405–439.

[66] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD Conference*. ACM, 1–12.

[67] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *SOSP*. ACM, 18–32.

[68] Thiemo Voigt and Bengt Ahlgren. 1999. Scheduling TCP in the Nemesis Operating System. In *Protocols for High-Speed Networks (IFIP Conference Proceedings, Vol. 158)*. Kluwer, 63–80.

[69] Gerhard Weikum and Gottfried Vossen. 2002. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, Chapter 5, 211–213.

[70] Charles B. Weinstock and William A. Wulf. 1988. An efficient algorithm for heap storage allocation. *ACM SIGPLAN Notices* 23, 10 (1988), 141–148.

[71] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *Proc. VLDB Endow.* 10, 7 (2017), 781–792.

[72] Ioannis Zarkadas, Tal Zussman, Jeremy Carin, Sheng Jiang, Yuhong Zhong, Jonas Pfefferle, Hubertus Franke, Junfeng Yang, Kostis Kaffes, Ryan Stutsman, and Asaf Cidon. 2023. BPF-oF: Storage Function Pushdown Over the Network. arXiv:2312.06808 [cs.OS]

[73] Nick Zavaritsky. 2024. We write our applications in ebpf: A Tale From a Telekom Operator. https://www.youtube.com/watch?v=uw8-BTn2p7M.

[74] 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 *OSDI*. USENIX Association, 375–393.

[75] Xinjing Zhou, Viktor Leis, Jinming Hu, Xiangyao Yu, and Michael Stonebraker. 2025. Practical DB-OS Co-Design with Privileged Kernel Bypass. *Proc. ACM Manag. Data* 3, 1 (2025).
[76] Yang Zhou, Xingyu Xiang, Matthew Kiley, Sowmya Dharanipragada, and Minlan Yu. 2024. DINT: Fast In-Kernel Distributed Transactions with eBPF. In *NSDI*. USENIX Association, 401–417.

Received October 2024; revised January 2025; accepted February 2025