# BENCHMARKING EMBEDDED KEY-VALUES STORES USING eBPF

**Sahakyan H.**
*Student, Institute for Informatics and Automation Problems of NAS RA, Yerevan, Armenia*
https://doi.org/10.5281/zenodo.15657638

**Abstract**

Embedded key-value stores run inside latency-critical paths, yet conventional benchmarks reveal only symptoms, not causes. We present ucsb-ebpf, a drop-in extension to the Unbranded Cloud Serving Benchmark that injects eBPF probes into every run. The tracer records per-thread syscall times and optional heap/slab events while adding under 1 % runtime overhead, so the same workload that measures throughput now pinpoints where each microsecond goes. Using seven standard mixes, we profile LevelDB, RocksDB, WiredTiger and LMDB on NVMe hardware. All code, data and analysis scripts are open sourced.

**Keywords**: eBPF; benchmarking; embedded key-value store; syscall tracing; memory profiling; performance analysis

## Introduction

Embedded key-value stores (KVSs) such as LevelDB, RocksDB, WiredTiger, and LMDB are increasingly embedded in latency-critical systems like load balancers, edge caches, and IoT gateways. This design eliminates a network hop by linking the entire database library directly into the application. Yet production traces still show abrupt p99 spikes when log-structured merge (LSM) compaction, allocator locks or fsync bursts collide with user traffic. External-only benchmarks cannot identify these internal causes; they can only report the symptom. This study argues that kernel-level eBPF tracing, combined with the Unbranded Cloud Serving Benchmark (UCSB), yields a more detailed accounting of how every microsecond is spent.

Over the decade since YCSB [1], benchmarking methods for embedded key-value stores have diversified: newer suites add hotspot mixes, transactional sequences, and SSD-oriented range scans, while micro-benchmarks isolate specific design choices. UCSB extends this trajectory by driving terabyte-scale data sets from a low-overhead C++ client. Yet every existing suite remains a black box—reporting throughput and latency but leaving the root causes of tail-latency spikes unexplained.

eBPF offers a non-intrusive path to causal insight. Small, verified eBPF probe programs can attach at runtime to any kernel or user-space symbol, sampling scheduler events and streaming data to buffers with minimal overhead. This approach was recently validated in studies tracing PostgreSQL and MySQL workloads [8]. By layering such probes on top of UCSB, we preserve the benchmark's external validity while turning it into a white-box diagnostic tool: the same workload that shows how fast a KVS runs now also reveals why it slows down.

This enhanced visibility has both practical and scientific value. Practitioners can decide whether tail spikes come from filesystem flushes, allocator mutexes or CPU scheduling, guiding fixes that cost only a configuration change. Researchers gain a reproducible, low-intrusion method for evaluating novel KVS designs, while reviewers can validate internal-cost claims instead of relying solely on aggregate throughput curves. In short, augmenting UCSB with eBPF closes the observability gap, converting a decade of black-box benchmarks into a causal performance microscope for embedded key-value stores.

This study makes four concrete contributions: (i) it delivers ucs-ebpf, a drop-in eBPF tracing layer for UCSB that adds < 1 % runtime overhead; (ii) it equips UCSB with optional heap/slab probes, enabling joint syscall- and memory-level accounting that existing suites lack; (iii) it applies this instrumentation to profile LevelDB, RocksDB, WiredTiger, and LMDB across seven standard workloads, pinpointing the dominant user- and kernel-space costs for each engine; and (iv) it releases all code, data, and analysis notebooks as open source, allowing researchers to reproduce and extend our findings.

The remainder of this paper is organized as follows: Section 2 reviews related work, Section 3 introduces the added eBPF segment, Section 4 presents the evaluation of four key value stores with harvested eBPF results, and Section 5 concludes the paper.

## Related work

Over the decade since YCSB [1], benchmarking methods for embedded KVSs have diversified. Newer suites such as YCSB++ [3] introduce hot-spot and eventual-consistency tests, and Transactional YCSB [4] adds ACID multi-operation mixes. Other tools like KVBench [5] enable flexible point vs. range query mixes for SSDs, while the micro-benchmarks in systems like KVell [6], WiscKey [7], PebblesDB [8], and RocksDB's db_bench target individual design choices. UCSB modernizes this lineage, scaling YCSB's model to terabyte-sized data and wider key/value distributions suitable for NVMe drives, while keeping the driver in C++ for minimal overhead.

Engine-specific micro-benchmarks complement general suites. RocksDB's db_bench, inherited from LevelDB, can sweep individual LSM phases. Research prototypes ship their own targeted workloads: KVell stresses per-core NVMe queues; WiscKey measures key/value separation on SSDs; PebblesDB quantifies write-amplification trade-offs. These tools expose fine-grained design choices, yet their instrumentation rarely extends beyond throughput and latency counters.

Beyond black-box metrics, several studies explore low-overhead tracing. Perf and DTrace offer coarse visibility, but eBPF has emerged as a lightweight alternative: vetted probe programs can time system calls, user-space functions and scheduler events with less than 1 % overhead. Recent explorations on PostgreSQL and MySQL confirm that eBPF captures detailed latency paths while perturbing workloads far less than in-process logging or ptrace-based tools. General-purpose toolkits such as BCC and bpftrace provide a reproducible substrate for such analyses.

To date, however, no study combines UCSB's large-scale, storage-aware workloads with eBPF instrumentation. Our work fills this gap, turning UCSB from a black-box scoreboard into a causal microscope that attributes tail-latency outliers to concrete kernel and user-space costs.

**Benchmark Architecture and eBPF Instrumentation**

In our benchmark, the Unbranded Cloud Serving Benchmark (UCSB) remains the workload driver. Its controller script, run.py, now can take two supplementary flags: --with-ebpf, which compiles our eBPF program, attaches the probe set, and spawns a helper thread for trace collection; and --with-ebpf-memory, which recompiles the probes with heap- and slab-allocation hooks. When the latter flag is omitted, the high-overhead memory kprobes are left out, holding total runtime overhead to ~1 %.

Probe groups

- Phase markers - uprobes on UCSB's timer_t::start/resume/pause/stop every workload phase for precise alignment with external metrics.
- Syscall tracing - raw-syscall tracepoints log per-thread counts and nanosecond latencies, streaming samples to buffers with sub-1 % overhead.
- Memory telemetry (optional) - uprobes on malloc/calloc/realloc/mmap together with kprobes on slab alloc/free record allocation size, stack trace, and lifetime; compiled in only when --with-ebpf-memory is specified.

A helper thread wakes every fifteen seconds, harvests phase timestamps, syscall info, thread life-cycle events, and when enabled memory summaries, then stores each snapshot as JSON in bench/ebpf/snapshots/.

To minimize measurement skew, we execute the benchmark twice per database workload combination:

- Timing pass - run with --with-ebpf alone to capture phase and syscall latencies at minimal overhead.
- Memory pass - repeat with --with-ebpf --with-ebpf-memory to gather allocation metrics.

We used this two-pass methodology for all results, and we recommend it for future studies. The first pass yields uncontaminated timing metrics, whereas the second (with memory tracking) provides heap usage insights without disturbing the already-recorded critical-path timings Thus, UCSB still reports how fast each engine runs, while the eBPF layer - applied judiciously in two stages - exposes where CPU time and memory are consumed, transforming a black-box benchmark into a causal microscope for embedded key-value stores.

The benchmark and tracer coordinate with a three-step UNIX-signal handshake. (i) When the runner finishes attaching the eBPF probes, it unblocks the suspended UCSB child by sending SIGUSR1 - this marks time zero for the run . (ii) At workload completion, the child notifies the tracer with SIGUSR2, the tracer's handler records a flag (SIGUSR2_received) so the harvesting loop knows the workload has ended and must flush the in-kernel buffers. (iii) After the tracer has drained all remaining events and written the final JSON snapshot, it sends a final SIGUSR1 back to the child, allowing the child process to exit cleanly and ensuring no late samples are lost during teardown. This bidirectional signalling guarantees that metrics collection starts only after every probe is running and stops only after every last event has been persisted.

Several studies [9-11] show that zipfian distribution is a good approximation for web data, so in our benchmark keys are selected from zipfian distribution with alpha = 0.99.

**Benchmark results**

We benchmarked the following workloads

- Write-Only: 100 % upserts or inserts (upserts) - stresses WAL throughput, memtable growth, and full LSM compaction path
- Read-Only: 100 % point reads - measures pure lookup latency
- Read-Heavy: 99 % reads / 1 % upserts - simulates rare state changes typical of session-stickiness tables
- Read-Mostly: 95 % reads / 5 % upserts - captures read-dominated loads with measurable write interference
- Balanced: 50 % reads / 50 % upserts - induces equal read/write pressure and worst-case mutex contention
- Range-Scan: 90 % short scans / 10 % point reads - exercises iterator logic, prefetching, and cache-line churn
- Remove-Only: 100 % deletes - probes tombstone creation, space reclamation, and compaction clean-up

Detailed parameters can be found in project repository's bench.json file under bench/workloads directory. The benchmark is run on machine with Intel Core i7-9750H CPUs (2.60 GHz), 16 GB RAM, Ubuntu 22.04, and kernel 6.5. Before each timing run, we drop system caches using /proc/sys/vm/drop_caches API.

For illustrative purposes, we grouped system calls into the following categories:

- Threading and Sync: Synchronization primitives and related syscalls such as set_robust_list
- Timing: Waiting, timing or sleeping (clock_nanosleep, clock_gettime etc.)
- Read I/O: read, pread64, readahead, lseek etc.
- Write I/O: write, pwrite64, writev, fdatasync etc.
- Memory Mapping and Allocations: mmap, munmap, mprotect, brk, madvise etc.

Note that in Fig. 1 we combine the CPU time of all four threads into a single value, because the system calls and user-space code execute in parallel.

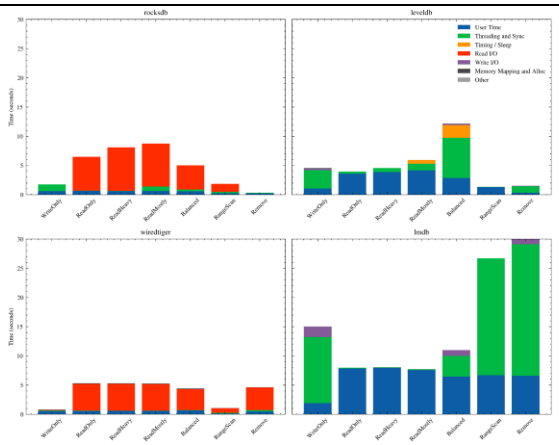


*Fig. 1. System Calls breakdown per Database in 4 threaded benchmark*

The shown figures (Fig.1-2) help make decisions without benchmarking on specific hardware. WiredTiger and RocksDB use heavy read system calls, therefore their performance is highly correlated with fast file systems, specifically fast reading hardware. LevelDB executes mainly in user space; its performance is therefore CPU-bound rather than I/O-bound. LMDB's heavy use of mutexes and related synchronization primitives introduces contention that limits scalability under multithreaded workloads.

Traditional benchmarks such as YCSB expose only aggregate throughput and latency, so they cannot attribute performance to user-space computation versus kernel-space I/O paths. Our extended UCSB framework adds per-syscall tracing, enabling the user-vs-kernel breakdown shown in Fig. 1–2.

The same decomposition can be applied to any additional key–value engine by deriving a new implementation from the ucsb::db_t abstract class. The system-call analysis is packaged as user-vs-kernel-time.ipynb Jupyter notebook in project repository.

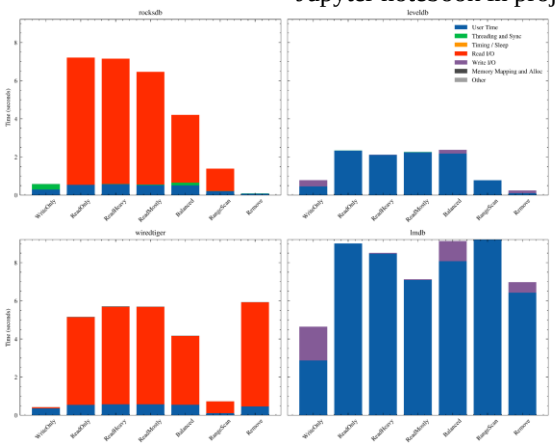


*Fig. 2. System Calls breakdown per Database in single threaded benchmark*

Beyond syscall latencies, the probes also build a fine-grained memory profile: every allocation and free is timestamped, live bytes are tallied in real time, and concurrent samples record both user-space RSS and kernel slab usage. Each 15 s snapshot is emitted to ebpf-memory-snapshots, ready for post-hoc correlation with workload outliers.

**Conclusion**

This paper introduced ucsb-ebpf, a lightweight eBPF add-on for the Unbranded Cloud Serving Benchmark that turns every run into a trace study while adding less than 1 % overhead [12]. With two command-line flags, researchers can collect per-thread syscall latencies and optional heap/slab snapshots without modifying benchmark code. We demonstrated its value by profiling four embedded stores—LevelDB, RocksDB, WiredTiger, and LMDB—under seven standard workloads on commodity NVMe hardware.

The breakdowns reveal different bottlenecks for each engine. RocksDB and WiredTiger spend most of their time inside read-oriented system calls, so faster storage would help more than extra CPU. LevelDB is almost entirely CPU-bound, indicating that parallelism or code-level optimizations should be the focus. LMDB scales poorly because a global mutex serializes critical paths; relaxing that lock is the clearest path to higher throughput.

These findings turn raw throughput figures into concrete guidance for operators and designers. Because ucsb-ebpf and all analysis scripts are open-sourced, the results are easy to reproduce, extend to new hardware, or apply to additional key–value engines.

**References:**

1. B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB", Proceedings of the 1st ACM Symposium on Cloud Computing, Association for Computing Machinery, New York, NY, USA, pp. 143–154, 2010.

2. J. Domaschka, S. Volpert, K. Maier, G. Eisenhart, and D. Seybold, “Using eBPF for Database Workload Tracing: An Explorative Study”, ACM /SPEC International Conference on Performance Engineering, Coimbra, Portugal, pp. 311–317, 2023.

3. S. Patil et al., “YCSB++: Benchmarking and Performance Debugging Advanced Features in Scalable Table Stores”, Carnegie Mellon University, 2011

4. A. Dey, A. Fekete, R. Nambiar, and U. Röhm, “YCSB+T: Benchmarking web-scale transactional databases” 2014 IEEE 30th International Conference on Data Engineering Workshops, pp. 223–230, 2014.

5. Z. Zhu, A. Saha, M. Athanassoulis, and S. Sarkar, “KVBench: A Key-Value Benchmarking Suite”, Proceedings of the Tenth International Workshop on Testing Database Systems, Association for Computing Machinery, New York, NY, USA, pp. 9–15, 2024.

6. B. Lepers, O. Balmau, K. Gupta, and W. Zwaenepoel, “Kvell+: Snapshot Isolation without Snapshots”, 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), USENIX Association, pp. 425–441, 2020.

7. L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “WiscKey: Separating Keys from Values in SSD-Conscious Storage”, ACM Trans. Storage, vol. 13, no. 1, 2017.

8. P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham, “PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees”, Proceedings of the 26th Symposium on Operating Systems Principles, Association for Computing Machinery, New York, NY, USA, pp. 497–514, 2017.

9. L. Breslau, P. Cue, L. Fan, G. Phillips, and S. Shenker, “Web Caching and Zipf-like Distributions: Evidence and Implications”, Proceedings IEEE INFOCOM, vol. 1, Oct. 1999.

10. B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, “Workload analysis of a large-scale key-value store”, SIGMETRICS Perform. Eval. Rev., vol. 40, no. 1, pp. 53–64, 2012.

11. B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky, “Small cache, big effect: provable load balancing for randomly partitioned cluster services” Proceedings of the 2nd ACM Symposium on Cloud Computing, Association for Computing Machinery, New York, NY, USA, 2011.

12. (2025) GitHub repository, H. Sahakyan, ucsb-ebpf, [Online]. Available: github.com/hov1417/ucsb-ebpf