Disaggregated RocksDB - Benchmarking and Performance Evaluation

Distributed Systems Lab Report

Duc Anh Phan d.a.phan@student.vu.nl
Storm Timmermans s.b.timmermans@student.vu.nl
Yuqi Yue y.yue@student.vu.nl

Krijn Doekemeijer
Supervisor
k.doekemeijer@vu.nl

Abstract

Key-value stores are fundamental to modern distributed systems, enabling high-performance data management for cloud computing and big data applications. This lab exercise extends RocksDB, a widely used key-value store, by introducing the ability to swap storage engines and evaluate their performance. By implementing POSIX and iouring storage engines and benchmarking them under local storage scenarios, this work aims to analyze the performance impact of hardware and software configurations, providing insight into optimal storage setups for diverse use cases.

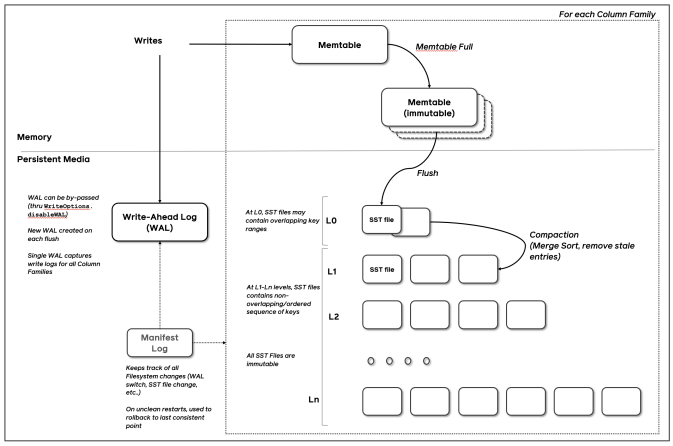


Figure 1: High-Level Architecture of RocksDB

1 Introduction

Efficient data storage and retrieval are critical to the performance of distributed systems. Key-value stores, such as RocksDB, have become integral to applications requiring high throughput and low latency. However, their performance is tightly coupled with the underlying storage engine, which mediates between the database and the file system. While RocksDB offers robust performance, its lack of support for swapping storage engines limits comparative analysis and optimal system configurations.

This project addresses this limitation by enabling Rocks DB to support multiple storage engines, including **POSIX**, **iouring** and **libaio** and exposing this functionality through a configurable interface. Furthermore, the project investigates the implications of disaggregated storage — an emerging paradigm where compute and storage resources are decoupled across separate nodes. This work benchmarks the performance of local versus remote storage to evaluate trade-offs in throughput, latency, and total cost of ownership.

Our contributions can be summarized as follows:
(1) We provide a pluggable design in RocksDB that simplifies swapping between `POSIX`, `libaio`, and `io_uring`, paving the way for future storage engine experimentation.
(2) We benchmark and profile these backends comprehensively in local and cluster SSD setups, highlighting how synchronous versus asynchronous calls and network latency can drastically affect performance metrics such as throughput, tail latency, and system call frequency.
(3) We analyze the overhead of tracing tools (`strace` vs. `perf`) to underscore the need for lightweight performance profiling.

The remainder of this paper is organized as follows. Section 3 details the functional and non-functional requirements that motivated our design. Section 4 describes how we integrated multiple storage engines into RocksDB via a new file-system backend layer and how we will perform the benchmarking. Section 5 covers key implementation details, especially the `FSWritableFile` and `FSRandomAccessFile` interfaces as well as how we set up the experiments in details. Section 6 presents a thorough experimental evaluation of performance, tracing overhead, and cluster-wide implications. Finally, Section 7 draws conclusions and discusses future directions such as fully asynchronous scheduling and advanced polling optimizations in RocksDB.

2 Background

2.1 Key-Value Stores and RocksDB

Key-value stores represent a simple yet powerful database paradigm where data is stored as key-value pairs. They are widely used in distributed systems because of their ability to handle large-scale data with low-latency operations. RocksDB, developed by Meta [4], is a high-performance embeddable key-value store optimized for fast storage environments, making it a popular choice for both industry and academia.

The following subsections is going to describe the basic operations of RocksDB as depicted in Figure 1.

2.1.1 Data Organization and Storage Model

RocksDB uses a Log-Structured Merge (LSM) Tree design, where data is written to an in-memory structure and periodically flushed to disk. This approach minimizes random writes to storage, favoring sequential writes instead. The key components include:

- **MemTable**: An in-memory data structure (usually a skip list or hash table) where writes are first stored.
- **Immutable MemTables**: Once the MemTable is full, it becomes immutable and is flushed to disk as a Sorted String Table (SSTable).
- **SSTables**: On-disk files storing sorted key-value pairs. SSTables are immutable and organized into levels to facilitate efficient lookups.

2.1.2 Write Path

All writes are first logged to the Write Ahead Log (WAL) for durability before being stored in the MemTable. This ensures data persistence in case of a crash. The write is then stored in the MemTable, allowing fast in-memory updates. When the MemTable is full, it is flushed to disk as an SSTable. This process involves sorting and compressing the data.

2.1.3 Read Path

When a key is queried, RocksDB searches for the key in the following order:

- MemTable (in-memory, fastest access).
- SSTables on disk, organized across levels.

Bloom Filters: RocksDB uses Bloom filters to optimize key lookups in SSTables, quickly determining if a key does not exist in a given SSTable.

2.1.4 Compaction

To manage storage and improve read performance, RocksDB performs compaction, a process of merging and reorganizing SSTables. SSTables are organized into levels (e.g., Level 0, Level 1, etc.), with increasing size and fewer files at higher levels. Data is compacted and merged as it moves to higher levels.

2.2 Storage Engines

The storage engine serves as the interface between the database and the file system, influencing performance metrics such as throughput and latency. Popular storage engines include POSIX, libaio, io_uring, and SPDK, each offering distinct trade-offs in terms of performance and complexity. POSIX is a traditional, widely-supported API, whereas iouring is a modern alternative designed to exploit asynchronous I/O operations for higher performance. In this assignment, we primarily focus on three storage engines:

- **POSIX I/O** is the standard and most basic interface for performing I/O on Linux systems. It provides synchronous (blocking) operations. The most common operations, like read() and write(), block the application until the operation completes.
- **libaio** provides asynchronous I/O capabilities, allowing an application to start multiple I/O operations without waiting for them to complete. It uses kernel support for asynchronous I/O
- **iouring** is the newest Linux I/O interface, designed to overcome limitations in existing asynchronous I/O frameworks like libaio. It provides a highly efficient mechanism for submitting and retrieving I/O operations using ring buffers, minimizing system calls by using shared memory ring buffers for submission and completion queues.

3 Requirements

3.1 Functional requirements

- Add a RocksDB file system backend that does a passthrough to the underlying file system
- Add functionality to the file-system backend to change the storage engine and incorporate it
- Add POSIX, libaio, and iouring as possible storage engine backends.
- Benchmark dbbench on top of the file-system backend and evaluate the impact of the storage engine in the db bench benchmark.

3.2 Non-functional requirements

- Benchmarks should be representative of real-world workloads, ensuring the results apply to practical scenarios.
- Benchmarks should evaluate not only fundamental metrics such as throughput and latency but also include the overhead of system calls.
- Results should provide insights into the trade-offs between local and disaggregated storage setups, helping in decision-making for diverse use cases.

4 Design

4.1 Extensible RocksDB

In this section, we present the high-level design of our extensible RocksDB, which supports swapping the storage backend while adhering to the functional and non-functional requirements outlined in Section 3.

The core of our design involves creating an interface that all RocksDB I/O operations—namely flushing, compaction, and reads—use. This simple design is illustrated in Figure 2.

Our swappable system forms the cornerstone of the extensible RocksDB design. Its purpose is to abstract I/O operations, ensuring that the upper layer of RocksDB interacts with the underlying storage system exclusively through this I/O interface. This modular approach enables RocksDB to remain flexible, adaptable, and optimized for diverse workloads. The backend provides a unified, transparent interface, ensuring seamless interaction with RocksDB regardless of the chosen storage engine.

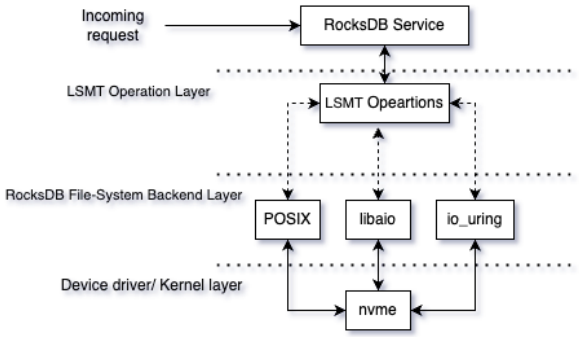


Figure 2: High Level Architecture of Extensible RocksDB

This abstract interface is implemented by various underlying storage backends, each using its method to interact with the system (e.g., Linux). Examples of these interactions include using popular mechanisms such as POSIX system calls, `libaio`, or `io_uring`.

Users can easily select and configure their preferred storage engine using one of the following methods:

- **Command-line option for `db_bench`**: Users can specify the storage backend via the `--fs_uri` flag.
- **RocksDB API**: Users can configure the desired backend in C++ using `DBOptions` with the `fs_uri` parameter.

4.2 Experiment Design

To evaluate the impact of different storage engines and configurations on RocksDB's performance in local storage setups, the initial focus is on static and fixed benchmarks. These benchmarks serve to validate the functionality of new file systems while also uncovering performance overhead differences among various backends. This structured approach ensures a stable foundation for subsequent experimentation.

Building on this, real-world applicability becomes a critical aspect of the design. Realistic workloads are essential for deriving meaningful insights, and this can be achieved by incorporating traces or employing standardized frameworks such as YCSB, which are designed to simulate real-world application scenarios effectively.

Furthermore, the design emphasizes the importance of using comprehensive metrics to thoroughly assess system performance. We begin by focusing on two key metrics—throughput and tail latency. Throughput measures how much data the system can process effectively within a given timeframe. High throughput in a database indicates the ability to handle a large number of requests or operations efficiently, which is critical in environments with continuous high-volume data transactions. Tail latency, on the other hand, refers to the small percentage of requests that take significantly longer to process than the average. While most requests may have low latency, these outliers can substantially impact overall system performance and user experience. Solely focusing on average latency can be misleading; therefore, we emphasize latency percentiles. Furthermore, we also include advanced metrics to provide a deeper understanding of system behavior and overhead, offering valuable insights into the trade-offs associated with different storage engine configurations. Those metrics including:

- Syscall counts and syscall time: The core differences between storage backends lie in how they interact with the system, such as their use of system calls. Monitoring these metrics helps identify performance differences between backends.
- CPU utilization: While a specific backend may deliver better performance, it might exhaust system resources. Understanding resource utilization across different backends informs trade-offs between performance and resource usage, enabling better system-level decisions.

5 Implementation

5.1 Extensible RocksDB

All write operations (e.g., writes to the Write Ahead Log (WAL) and SSTables) utilize the `FSWritableFile` interface. Meanwhile, read operations to metadata files such as the OPTIONS file, MANIFEST file, and CURRENT file using the `FSSequentialFile` interface. Reads to SSTables are handled using the `FSRandomAccessFile` interface, designed for random access patterns in data retrieval.

Given the central role of these interfaces, our focus is on implementing two key components: `FSWritableFile` and `FSRandomAccessFile`. These components are critical for RocksDB's I/O operations.

- `FSWritableFile`: The two primary methods to implement are `Write` and `WritePositioned`. Each backend storage system requires its specific implementation:
  - `LibaioWritableFile`: Implements `Write` and `WritePositioned` using asynchronous I/O system calls provided by `libaio`.
  - `IOUringWritableFile`: Leverages `io_uring` to handle asynchronous writes with minimal overhead through submission and completion queues.
  - `PosixWritableFile`: The default implementation in RocksDB, which uses traditional synchronous POSIX system calls.

  For each method, the system calls (e.g., `write()`, `pwrite()`) are adapted to suit the specific characteristics of the backend storage engine.
- `FSRandomAccessFile`: The primary method to implement is `Read`. Similar to `FSWritableFile`, this implementation is tailored for each backend:
  - `LibaioRandomAccessFile`: Uses `libaio` system calls to perform asynchronous read operations.
  - `IOUringRandomAccessFile`: Utilizes `io_uring` for high-performance asynchronous read access.

All methods discussed (`Read`, `Write`, `WritePositioned`) are designed to execute synchronously, ensuring that the operations return results immediately without deferring execution. To achieve this, synchronous I/O operations in `libaio` rely on `io_getevents` to wait for completion, while `io_uring` operations use `io_uring_wait_cqe` for polling or waiting for completion.

At a higher level, these file interfaces are managed by the `FileSystem` interface, which oversees all file operations and serves as the main interface for RocksDB to interact with the underlying file system. The `FileSystem` interface is modular, allowing the integration of different backends:

- `IOUringFileSystem`: Implements the file system using `io_uring` for all read and write operations.
- `LibaioFileSystem`: Provides an implementation based on `libaio`.
- `PosixFileSystem`: Uses the default POSIX interface for traditional synchronous I/O.

To support flexibility and experimentation, we implement this file system layer as a plugin for RocksDB. Users can specify the desired backend storage system using the `fs_uri` option. For example, setting `fs_uri://libaio` configures RocksDB to use the `LibaioFileSystem`, while `fs_uri://iouring` selects the `IOUringFileSystem`. This design simplifies the integration process and allows developers to easily switch between backends based on their workload requirements.

Overall, this approach provides a robust, extensible I/O subsystem for RocksDB, capable of supporting modern storage solutions while remaining backward compatible with traditional POSIX-based systems.

We next experiment with the backends on a real cluster provided by our supervisor. Another emerging trend in storage is disaggregated storage, where compute and storage nodes are physically separate. In our setup, we will deploy remote SSDs and integrate them into the cluster, allowing us to compare performance between locally attached SSDs and remote SSDs.

5.2 Experiment Setup

5.2.1 Static db_bench

We first utilize few popular db_bench benchmarks - to test the operations of the plugin as well as to gain few insights into the performance of different backends. We choose fixed-size (key, value) pairs (16-byte keys and 1000-byte values) and reserve only 2 MiB for the write buffer. We perform the benchmark with direct I/O to eliminate the effects of page-based cache. Two metrics we observe through this benchmark are throughput and latency.

5.2.2 Real-world workload

Researchers usually consider the workloads generated by YCSB to closely resemble real-world workloads. YCSB can generate queries with statistical properties similar to those of realistic workloads, including query type ratios, KV-pair hotness distribution, and value size distribution.

However, the observed results reveal discrepancies between YCSB-generated workloads and replayed workloads:

- The number of block reads from YCSB is at least 7.7x that of the replayed results, and the amount of read-bytes is approximately 6.2x. These results indicate an extremely high read amplification.
- The number of block cache hits in YCSB workloads is only about 0.17x that of the replayed results.

To better emulate real-world workloads, [2] proposes a key-range based model. In this model, the entire keyspace is partitioned into several smaller key-ranges. Instead of modeling KV-pair accesses based on global keyspace statistics, this approach focuses on the hotness of individual key-ranges. By doing so, the generated workloads more accurately reflect the access patterns observed in realistic scenarios. [2] developed a benchmark called "mixgraph" in db_bench, which can use the four sets of parameters to generate the synthetic workload. The workload is statistically similar to the original one.

We used the parameters below to generate the workload that simulates the queries of ZippyDB described in [2].

```
db_bench --benchmarks="mixgraph" --fs_uri=efs://
    libaio -
    use_direct_io_for_flush_and_compaction=true
    -use_direct_reads=true -cache_size=268435456
     -keyrange_dist_a=14.18 -keyrange_dist_b
    =-2.917 -keyrange_dist_c=0.0164 -
    keyrange_dist_d=-0.08082 -keyrange_num=30 -
    value_k=0.2615 -value_sigma=25.45 -iter_k
    =2.517 -iter_sigma=14.236 -mix_get_ratio
    =0.85 -mix_put_ratio=0.14 -mix_seek_ratio
    =0.01 -sine_mix_rate_interval_milliseconds
    =5000 -sine_a=1000 -sine_b=0.000073 -sine_d
    =4500 --perf_level=2 -reads=8400000 -num
    =1000000 -key_size=48
```

5.2.3 Strace

strace [1] is a powerful diagnostic, debugging, and monitoring tool in Linux that is widely used to trace and analyze system calls and signals executed by a program. A system call (commonly referred to as a syscall) serves as the interface between user-space applications and the Linux kernel. Through syscalls, user-space programs can request services from the operating system, such as performing file I/O operations, managing processes, allocating memory, or handling network communication.

By intercepting and logging syscalls made by a program, **strace** provides developers and system administrators with a detailed view of how the program interacts with the operating system. This insight can help identify performance bottlenecks, debug system-level errors, and optimize resource usage. Additionally, **strace** can log the arguments passed to syscalls, their return values, and the duration of each syscall, offering valuable information for performance analysis.

In this assignment, we leverage the capabilities of **strace** to evaluate the behavior of `db_bench` across different storage backends. Specifically, **strace** is used to count the number of syscalls invoked by `db_bench` while operating through supported backends such as `POSIX`, `io_uring`, and `libaio`. This information is critical for understanding the system's behavior under different configurations, as the frequency and type of syscalls can significantly influence performance.

Furthermore, we measure the latency of each syscall using **strace**. Syscall latency represents the time taken for the kernel to execute the requested operation and return control to the application. By analyzing this data, we can interpret the performance differences between backends, identifying the impact of various factors such as I/O scheduling, context switching, and kernel overhead. This analysis plays a key role in assessing the efficiency of each

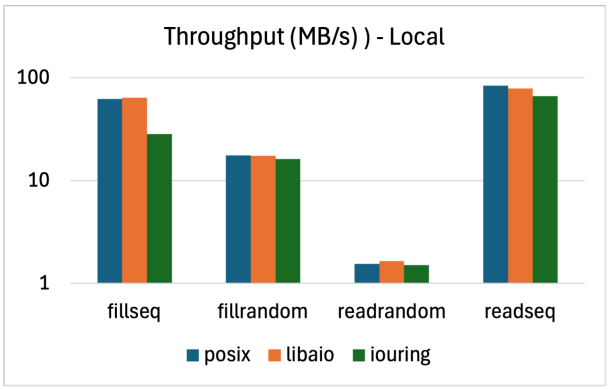


Figure 3: Throughput - local benchmark

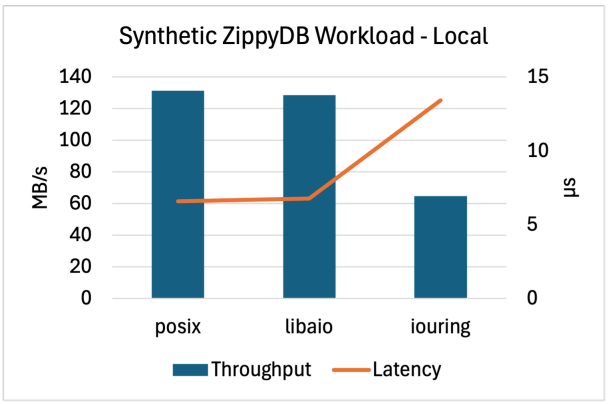


Figure 4: ZippyDB workload - local benchmark

storage backend and its suitability for `db_bench`'s workload.

5.2.4 perf

perf [3] is a powerful performance analysis and profiling tool built into the Linux kernel. It enables developers, system administrators, and performance engineers to monitor and analyze the performance characteristics of applications, processes, or the entire system. By leveraging hardware performance counters, software events, and tracepoints, perf provides detailed insights into how a system or application uses its resources.

strace incurs higher overhead because it operates by attaching to a process using the ptrace system call, intercepting and monitoring every system call made by the traced process. Each intercepted syscall requires a context switch between the traced process and strace, which introduces significant overhead, especially for programs with frequent system calls. Additionally, strace logs detailed information for each syscall, including its name, arguments, return value, and execution time. Writing this data to stdout or a file further amplifies the overhead, particularly for syscall-intensive applications. In contrast, perf uses kernel-level instrumentation and direct access to hardware performance counters, enabling it to collect aggregated performance data with minimal interference. By avoiding frequent context switches and detailed logging for each event, perf maintains a much lower overhead compared to strace.

5.2.5 Cluster setup

In this experiment, we used a cluster that provides two configurations—virtual SSD and remote SSD. We compiled our code on the remote cluster and set up the experiment using the provided script.

6 Results and Discussion

6.1 Local Benchmark

In our local SSD throuhput experiments (Figure 3), *fillseq* throughput for `posix` and `libaio` remains around 62–64 MB/s, whereas `io_uring` stands lower at approximately 28 MB/s. All three engines perform more closely under *fillrandom*, near 16–17 MB/s. They also converge in *readrandom* at about 1.5–1.6 MB/s, suggesting no clear advantage for any interface in random reads. Finally, *readseq* shows `posix` leading at 84 MB/s, followed by `libaio` at 78 MB/s and `io_uring` at 66 MB/s, indicating that `io_uring` generally trails the more mature engines in sequential workloads.

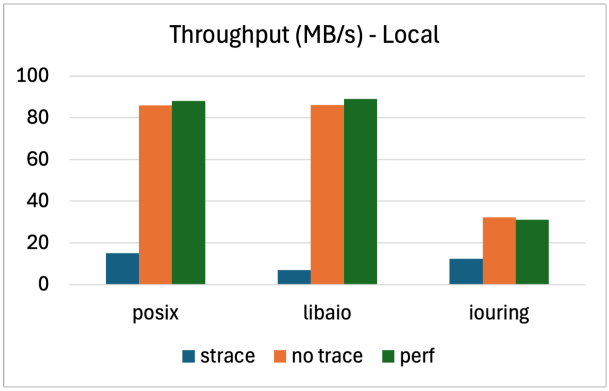
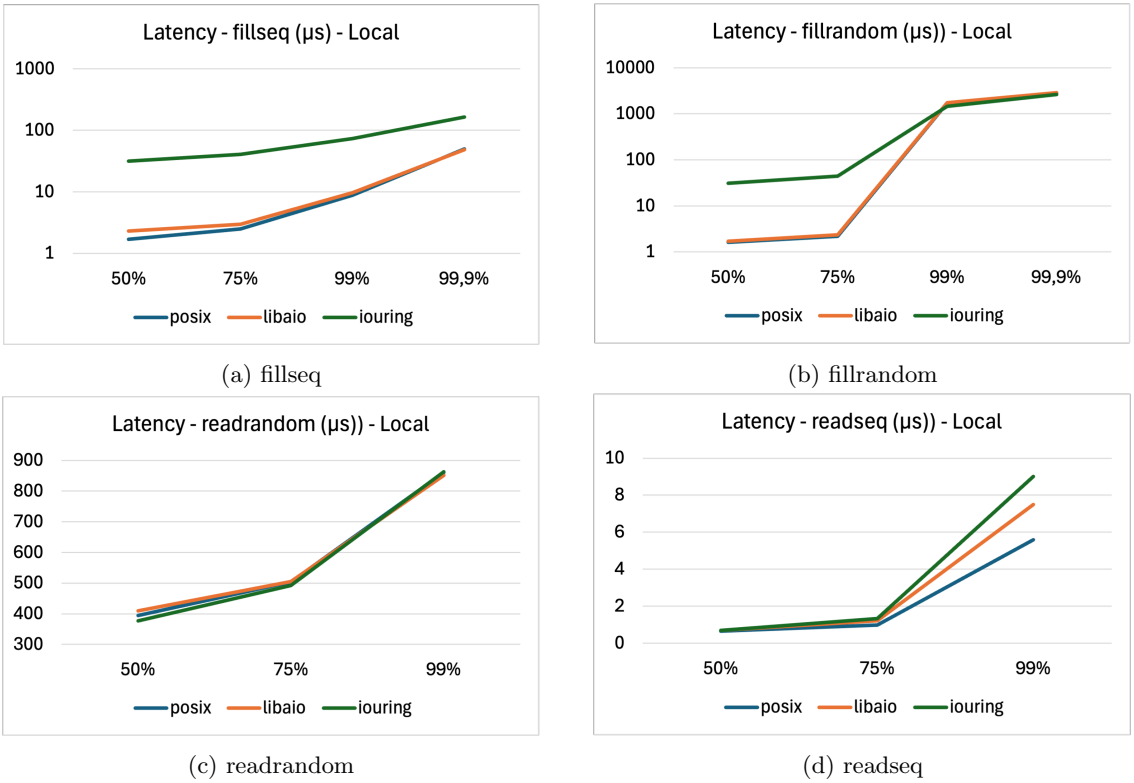


Figure 5: Tracing tool impact - local benchmark

Latency results (Figure 6) highlight that `posix` and `libaio` maintain lower microsecond-level latencies under *fillseq* and *fillrandom*, with `io_uring` exhibiting tens of microseconds at lower percentiles and rising more sharply at the tail. For *readrandom*, all three settle in the hundreds of microseconds at the median, diverging moderately at the 99% and 99.9% levels. Under *readseq*, median latencies are sub–2 µs for every engine, though `io_uring` again shows the largest spikes at the tail. Overall, `posix` and `libaio` remain more stable across percentiles, whereas `io_uring` demands additional tuning or higher concurrency to match or surpass their performance fully.

In the synthetic ZippyDB workload (Figure 4), `POSIX` tops out at approximately 131 MB/s with a latency of around 6.6 µs, closely followed by `libaio` at 128 MB/s and 6.8 µs. In contrast, `io_uring` provides noticeably lower throughput, at about 64 MB/s, and has a higher latency of roughly 13.4 µs. Overall, `POSIX` and `libaio` display comparable performance in both metrics, while `io_uring` lags significantly in throughput and exhibits higher latency.

To study the impact of different functions on throughput and latency, we measured the time spent on system calls using both strace and perf. Figure 5 compares the throughput of db_bench with either tracing tool or not. With `strace` enabled, throughput plummets significantly across all three backends, reflecting the overhead of intercepting and logging each system call. By contrast, using `perf` introduces only a modest performance hit compared



(a) fillseq

(b) fillrandom

(c) readrandom

(d) readseq

Figure 6: Latency - local benchmark

to no tracing at all. For example, `POSIX` and `libaio` each exceed 80 MB/s without tracing but drop to around 10 MB/s when `strace` is active; with `perf`, their throughput remains in roughly the same high range as the "no trace" scenario. `io_uring` also suffers more from `strace` than from `perf`, though its overall throughput stays lower than `POSIX` and `libaio` under all conditions. These results confirm that `strace` imposes a substantially greater overhead on `db_bench` measurements than `perf`, making `perf` the more efficient tool for gathering profiling data without severely distorting performance outcomes.

Therefore, we decide to use perf to analyze the impact of different functions on the overall performance. Using `perf` to analyze the three engines (POSIX, libaio, and `io_uring`) reveals that their respective system consumes most execution time calls—`read` and `write` for POSIX, `io_getevents` and `io_submit` for libaio, and `submit` and `get completion queue` for `io_uring` (see Figures 7, 8, and 9). Although libaio and `io_uring` both support asynchronous operations, the current RocksDB codebase lacks a mechanism to exploit true latency hiding; simple implementations of these engines still end up blocking until operations are complete. In theory, this negates any immediate advantage of switching from POSIX to an asynchronous engine. However, `io_uring` has the potential to reduce the overall number of system calls, thereby improving latency and throughput if implemented with the full polling mode. Incorporating such a mechanism would require significant re-engineering of RocksDB to eliminate blocking paths, so we defer this work to the future.

Overall, our local SSD experiments confirm that `POSIX` and `libaio` consistently deliver higher throughput and more stable latencies, while `io_uring` lags behind unless further tuned. Tracing with `strace` introduces a significant performance penalty, whereas `perf` proves more efficient for gathering profiling data. Analyzing these engines reveals that blocking paths in RocksDB diminish the advantages of asynchronous I/O, although `io_uring` remains promising if integrated with a non-blocking design. Having established these baseline insights, we now move on to real-world cluster benchmarking to evaluate whether the observed patterns hold at a larger scale and in production-like environments.

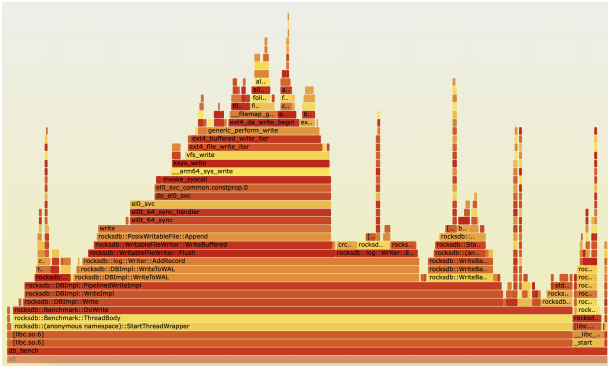


Figure 7: Posix - local benchmark

6.2 Cluster Benchmark

Our cluster throughput experiments evaluated virtual and remote SSD setups using the same benchmarks as in the local tests. For the *fillseq* workload, in the virtual setup (Figure 10), `libaio` achieves the highest throughput at 240.5 MB/s, followed closely by `POSIX` with 219 MB/s. However, `io_uring` lags far behind with only 15.8 MB/s. In contrast, for the remote setup (Figure 10), both `POSIX` and `libaio` show a noticeable drop in throughput to 166.7 MB/s and 159 MB/s, respectively, while `io_uring` continues to deliver low performance at 13.8 MB/s.

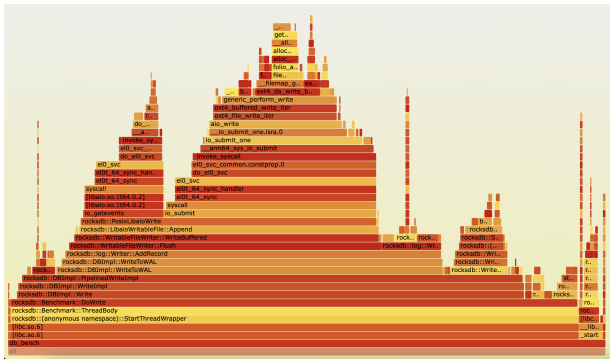


Figure 8: Libaio - local benchmark

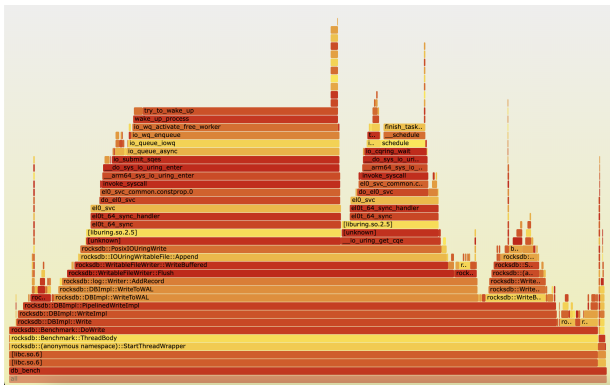


Figure 9: iouring - local benchmark

For the *readrandom* workload, the performance across backends in the virtual setup is relatively close, with throughput ranging from 81.8 MB/s for `libaio` to 89.1 MB/s for `POSIX`. Interestingly, in the remote setup, `io_uring` outperforms the other two backends, achieving 91.3 MB/s compared to 79.6 MB/s for `libaio` and 50.8 MB/s for `POSIX`. This result suggests that `io_uring` can handle random reads more efficiently in remote scenarios, likely due to its lower syscall overhead.

For the *readseq* workload, `POSIX` achieves the highest throughput in the virtual setup, reaching 903.6 MB/s. `libaio` and `io_uring` follow with 759.8 MB/s and 846.4 MB/s, respectively. In the remote setup, `io_uring` surprisingly achieves the highest throughput at 861.1 MB/s, surpassing both `libaio` (711.5 MB/s) and `POSIX` (557.2 MB/s). This result highlights the potential of `io_uring` for handling large sequential reads in remote environments, potentially due to its ability to reduce syscall latency through polling mechanisms.

As shown in Figure 11, comparing the virtual and remote setups reveals several key trends. For *fillseq*, both `POSIX` and `libaio` experience a significant drop in throughput (approximately 25%) in the remote setup compared to the virtual one, while `io_uring` exhibits minimal change. This minimal drop for `io_uring` reflects its lower initial performance in the virtual environment. In *readseq*, the throughput for `POSIX` and `libaio` decreases by 38% and 6%, respectively, in the remote setup, whereas `io_uring` demonstrates a slight increase in throughput. Lastly, in *readrandom*, `POSIX` shows a noticeable drop in throughput (around 43%) in the remote setup compared to the virtual one, while `libaio` remains relatively stable with only a slight decrease. Interestingly, `io_uring` demonstrates a slight increase in throughput (around 4%) in the remote setup, outperforming both `POSIX` and `libaio` in this workload.

The results highlight the impact of network-induced latency on throughput, with `POSIX` and `libaio` showing drops in write-intensive and sequential workloads in the remote setup. `io_uring`, while underperforming in the virtual setup, shows potential advantages in handling remote storage, especially in read-heavy workloads, due to its reduced syscall overhead and asynchronous nature.

Latency results in the virtual cluster benchmark (Figure 12) indicate that for the *fillseq* workload, `POSIX` and `libaio` exhibit similar latency behavior across all percentiles, with a gradual increase from 1.63 µs and 1.53 µs at the 50% percentile to 13.42 µs and 9.99 µs at the 99.9% percentile, respectively. In contrast, `io_uring` maintains significantly higher latency across all percentiles, starting at 31.06 µs at the 50% percentile and rising to 75.99 µs at the 99.9% percentile, highlighting its poorer performance in sequential write workloads.

For the *readrandom* workload, all three backends exhibit similar and stable latency across all percentiles, with median latencies ranging from 5.11 µs to 5.51 µs. Even at the 99% and 99.9% percentiles, the differences remain minimal, with latencies around 10 µs and 20 µs, respectively, indicating that none of the backends faces significant performance degradation under random read workloads in the virtual setup.

In the *readseq* workload, all three backends show comparable and stable performance up to the 75% percentile, with median latencies of 0.58 µs. Beyond the 75% percentile, latencies increase, with `libaio` showing the highest rise to 8.03 µs at the 99.9% percentile, followed by `io_uring` at 6.31 µs and `POSIX` at 5.94 µs. This indicates that while the three backends handle sequential reads well at lower percentiles, `libaio` experiences slightly higher tail latency under high contention.

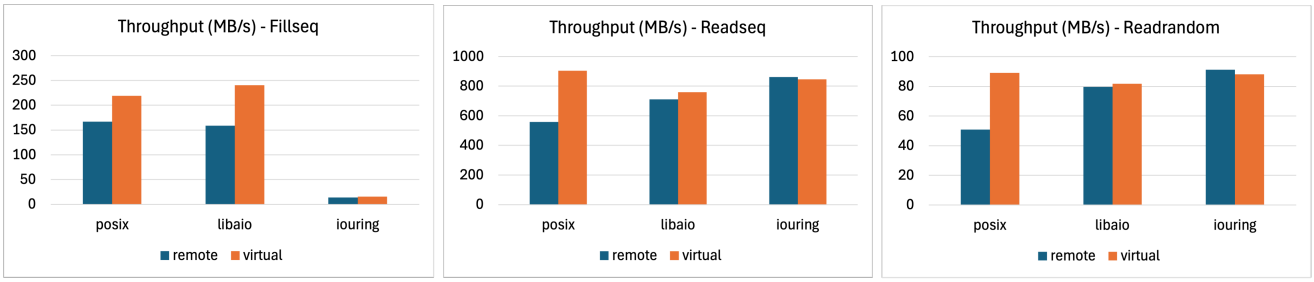
In the remote cluster benchmark (Figure 13), *fillseq* results show that while `POSIX` and `libaio` maintain relatively low latencies at the 50% and 75% percentiles, with values ranging from 1.85 µs to 2.8 µs, they exhibit noticeable increases at higher percentiles. At the 99.9% percentile, `POSIX` and `libaio` reach 17.43 µs and 17.75 µs, respectively. Meanwhile, `io_uring` shows consistently higher latency across all percentiles, starting at 34.94 µs at the 50% percentile and increasing sharply to 115.13 µs at the 99.9% percentile, indicating its poorer performance under sequential write-intensive workloads in remote setups.

For the *readrandom* workload, `POSIX` exhibits a dramatic rise in latency at higher percentiles, reaching 164.97 µs at the 99% percentile and 242.49 µs at the 99.9% percentile. In contrast, both `libaio` and `io_uring` show relatively stable performance, with their 99.9% percentile latencies remaining under 21 µs. Notably, `io_uring` delivers slightly lower latency than `libaio` at the higher percentiles, indicating its potential advantage in handling random reads in remote environments.

For the *readseq* workload, all three backends perform similarly up to the 75% percentile, with median latencies of 0.58 µs. However, beyond the 75% percentile, `POSIX` experiences a steeper increase, reaching 14.46 µs at the 99.9% percentile, while `libaio` and `io_uring` maintain lower tail latencies of 9.62 µs and 8.38 µs, respectively. This suggests that while all backends handle sequential reads well at lower contention levels, `POSIX` suffers from

(a) virtual (b) remote

Figure 10: Throughput - virtual & remote cluster benchmark



(a) fillseq (b) readseq (c) readrandom

Figure 11: Throughput - virtual *vs* remote cluster benchmark

higher tail latency under heavy contention in remote setups.

In both virtual and remote cluster benchmarks, *fillseq* exhibits a similar trend, with `POSIX` and `libaio` maintaining lower latencies across percentiles compared to `io_uring`, which consistently shows higher latency, particularly at the tail. However, the trends for *readrandom* and *readseq* differ between the two setups. In the virtual setup, all three backends show stable latency for *readrandom*, while in the remote setup, `POSIX` exhibits a significant rise in tail latency, whereas `libaio` and `io_uring` remain relatively stable. For *readseq*, the virtual setup shows an increase in tail latency for `libaio`, while in the remote setup, `POSIX` experiences a noticeable increase in tail latency.

Overall, the cluster benchmark results demonstrate distinct throughput and latency behaviors in virtual and remote SSD setups, highlighting the influence of workload types and backend characteristics. For throughput, *readseq* consistently outperforms other workloads across all backends in both virtual and remote setups. While `POSIX` and `libaio` show better performance in *fillseq* than in *readrandom*, `io_uring` presents an inverse trend, with relatively low throughput in *fillseq* but slightly higher performance in *readrandom*, particularly in the remote setup where it even surpasses both `POSIX` and `libaio`. These trends suggest that `io_uring` demonstrates improved adaptability for random read-heavy workloads, especially in high-latency environments. In terms of latency, *fillseq* exhibits similar trends across virtual and remote setups, with `POSIX` and `libaio` maintaining lower latency than `io_uring`. However, *readrandom* and *readseq* reveal differing patterns: in the virtual setup, all three backends show stable performance in *readrandom*, whereas in the remote setup, `POSIX` faces significant tail latency increases, while `libaio` and `io_uring` remain stable. For *readseq*, `libaio` exhibits higher tail latency in the virtual setup, while `POSIX` shows greater tail latency in the remote setup. These results indicate that while `POSIX` and `libaio` excel in low-latency environments, `io_uring` proves more resilient under high-latency conditions, especially for random reads.

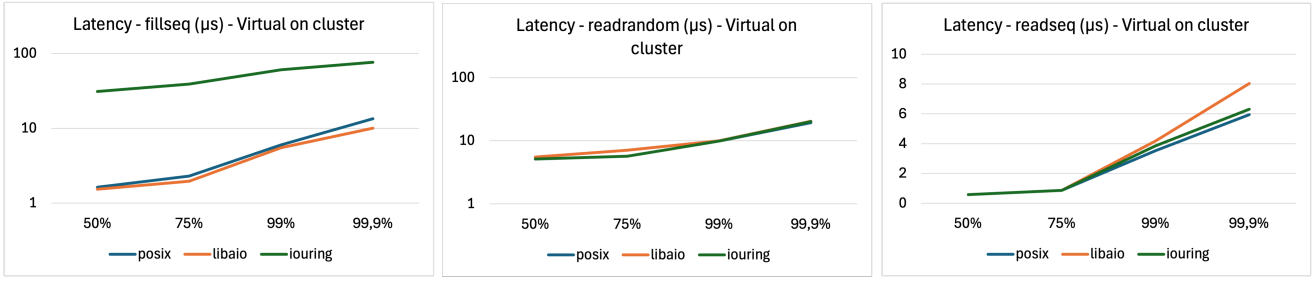
7 Conclusion

In this work, we designed and implemented an extensible RocksDB framework capable of seamlessly swapping out the underlying storage engine. By encapsulating I/O operations behind a common file interface, our design allowed RocksDB to support POSIX, `libaio`, and `io_uring` backends. We integrated these backends into `db_bench` and extensively profiled their performance on both local and cluster (virtual and remote SSD) setups under different workloads, including synthetic (*fillseq*, *fillrandom*, *readrandom*, *readseq*) and more realistic benchmarks such as a ZippyDB-like workload.

Key findings from our evaluation include:

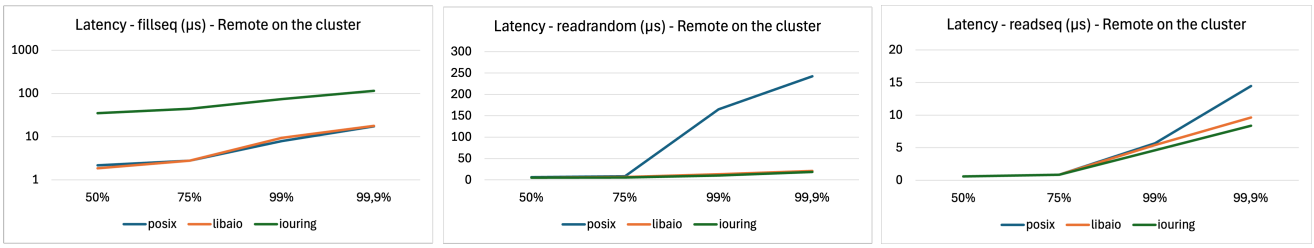
- **Local SSD Performance.** On local SSDs, `POSIX` and `libaio` generally offered higher throughput and lower latency than `io_uring` in both sequential and random access patterns. In particular, `posix` and `libaio` showed similar performance in sequential writes (*fillseq*) and reads (*readseq*). `io_uring` lagged behind, especially in sequential workloads.

- **Cluster SSD Performance.** In more realistic environments (virtualized SSDs and remote SSDs), we observed more nuanced behaviors. While `POSIX`



(a) fillseq (b) readrandom (c) readseq

Figure 12: Latency - virtual cluster benchmark



(a) fillseq (b) readrandom (c) readseq

Figure 13: Latency - remote cluster benchmark

and `libaio` still excelled in write-intensive and sequential workloads, `io_uring` closed the gap—and in some remote scenarios even outperformed the other backends—in read-heavy workloads. This improvement seems tied to `io_uring`'s ability to reduce syscall overhead when latency is primarily network-driven.

- **Latency & Tail Behavior.** Across setups, `posix` and `libaio` often maintained lower median latencies, but exhibited higher tail-latency spikes in certain cluster workloads. Meanwhile, `io_uring` sometimes struggled at the local setup but displayed promising tail-latency characteristics for remote, random read-intensive workloads.

- **Tracing Overheads.** Our comparison of `strace` and `perf` for profiling underscored that `strace` introduces a severe performance penalty—throughput plummets by as much as an order of magnitude due to frequent context switches and detailed syscall logging. `perf`, on the other hand, incurred significantly lower overhead, making it more suitable for real-time performance analysis of RocksDB.

Lessons learned. First, integrating asynchronous I/O interfaces (like `libaio` and `io_uring`) into RocksDB does not automatically deliver performance gains if the upper layers still block on I/O completions. A key insight is that fully exploiting asynchronous capabilities requires rethinking the way RocksDB schedules and pipelines I/O operations. Second, while `posix` often provides competitive performance in low-latency environments, `io_uring` offers considerable promise in higher-latency environments once tuned to run in an asynchronous or polled mode. Finally, tracing and debugging methodologies should carefully account for overheads: `strace` can drastically distort results, whereas kernel-level sampling via `perf` provides deeper insights with minimal disruption.

Future work will focus on:

- **Non-blocking Integration.** Extending RocksDB's design to better pipeline reads and writes, reducing blocking paths so that backends like `libaio` and `io_uring` can fully hide latencies and exploit parallelism.

- **Tuning and Polling Modes.** Investigating `io_uring`'s submission and completion polling for higher throughput and more stable tail latency.

- **Production-like Traces.** Incorporating extensive real-world traces (beyond YCSB and synthetic workloads) to capture nuanced access patterns, caching behaviors, and scheduling artifacts in production deployments.

Overall, this work demonstrates the viability and potential of building an extensible I/O subsystem in RocksDB to leverage modern asynchronous interfaces. While traditional `posix` I/O remains surprisingly competitive, `libaio` and `io_uring` can outshine it under specific workload characteristics and latencies, particularly with future enhancements to RocksDB's internal design that capitalize on genuine asynchronous operations.

References

[1] strace. https://strace.io/. Accessed: 2025-01-15.

[2] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies*, FAST'20, page 209–224, USA, 2020. USENIX Association.

[3] Arnaldo Carvalho de Melo and Red Hat. The new linux ' perf ' tools. 2010.

[4] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Rocksdb: Evolution of development

priorities in a key-value store serving large-scale applications. *ACM Trans. Storage*, 17(4), October 2021.

A Time sheets

Table 1: Time Scheduling

Activity	**Time spent (hours)**
think-time	50
dev-time	30
xp-time	75
analysis-time	45
write-time	50
wasted-time	10
total-time	**260**

Github: https://github.com/anhphantq/efs.git