Engineswap: RocksDB with io_uring and its performance

Joachim Bose jbe312, Alexis Dragoste ade204, Martin Karsai mki333

January 15, 2025

Abstract

RocksDB is a high-performance embedded Key-Value store and a critical piece of computing infrastructure where throughput and latency are important, which includes CockroachDB and Cassandra . Recent work in storage space has resulted in the development of io_uring, leading us to an opportunity to attempt to improve understanding and or performance of RocksDB and io_uring by combining the two. We designed and implemented engineswap, a RocksDB plugin able to switch storage API between io_uring, posix and libaio. When evaluating engineswap we found io_uring to have worse throughput and latency compared to the other engines. Additionally, we found the file system latency to increase when comparing posix and io_uring.

Contents

1 Introduction

Key-Value (KV) stores are programs able to store large amounts of data into files on a file-system and retrieve the data by searching for its associated key. Such KV stores like RocksDB [1] have been used in many other database projects like MySQL [2], MongoDB [3] and CockroachDB [4], themselves often used in many applications including web applications [5] and control planes [6], positioning themselves in the core of computing infrastructure. Therefore, efforts to improve KV stores like RocksDB may be helpful to many projects.

Rocksdb uses the Posix system calls to interface with storage devices (I/O) because it is highly supported across many Operating systems. However, recent developments have built new interfaces for I/O including libaio and io_uring, which have shown promising performance in previous studies [7].

This project aims to understand performance differences between the io_uring, posix and libaio interfaces by extending the capabilities of RocksDB to support configurable storage engines. Specifically, the goal is to integrate the ability to dynamically switch between storage engines, such as POSIX, io_uring, and potentially libaio, within RocksDB's file system backend. This functionality will allow end-users to configure storage engines through a simple interface and evaluate their impact on database performance.

Additionally, this project presents the design and implementation of experimental benchmarks to measure performance across various configurations. RocksDB's built-in benchmarking tool, 'db_bench', will serve as the primary platform for testing. The benchmarks will assess factors such as key-value pair size and workload intensity.

This report brings the following contributions:

- We design a RocksDB plugin enabling a switch between using io_uring and posix API's. The design and its motivation is explained in section 3.
- We implement engineswap and a system around it. Additionally, we published a fork from RocksDB with the engineswap plugin to GitHub:
  https://github.com/JoachimBose/rocksdb-VE
  The implementation is explained in section 4.
- We Evaluate the use of io_uring with and without submission queue polling, posix and libaio for RocksDB using engineswap and find that throughput and latency with io_uring are slightly worse with libaio, worse with plain io_uring and much worse when io_uring submission queue polling is enabled. Additionally, we find the file system latency to be the culprit of the increase in latency in io_uring. This evaluation is in section 5.

The rest of this report is structured by providing some background in section 2 about RocksDB and storage APIs, followed up by section 3 where we design engineswap. Next, we implement engineswap in section 4 and use it for experiments in chapter 5.

2 Background

Key-value (KV) stores have become fundamental components of modern computing systems, enabling efficient storage and retrieval of data through the use of simple key-value pairs. Key-value pairs can be inserted using a put operations and retrieved with a get operation. These systems are widely used in cloud-based applications, big data processing, etc. , where performance in terms of latency, throughput, and scalability is critical.

2.1 RocksDB

Our work is concerned with, RocksDB [1], a KV store developed by Meta. It is built on the Log-Structured Merge (LSM) tree, a data structure that provides efficient write and read operations by organizing data into sequentially written immutable files called SSTables (Sorted String Tables). When data is inserted or updated, it is first written to an in-memory write-ahead log (WAL) to ensure durability, followed by insertion into an in-memory MemTable. Once the MemTable reaches its capacity, it is flushed to disk as an SSTable. This hierarchical organization helps RocksDB achieve low-latency writes while optimizing storage utilization [8].

The core operation of the LSM tree in RocksDB involves periodic compaction, a process in which smaller SSTables are merged into larger ones to reduce fragmentation and ensure efficient reads. Compaction minimizes the number of SSTables that need to be queried during data retrieval, effectively balancing write amplification and read amplification.

2.2 Storage APIs

Each KV store uses a storage engine, which serves as the intermediary between the database system and the underlying file system and hardware, in Rocksdb's case this is just a conversion between Rocksdb's internal file system API and the posix API. This posix API provides write() and read() synchronous system calls, usually through the kernel to the file system. However, emerging storage technologies like io_uring have introduced opportunities for improved I/O efficiency, especially in high-performance applications [9]. In this work, we focus mainly on io_uring and posix, due to their promising performance and relevance, sometimes also libaio for additional context.

2.2.1 Posix I/O

posix is the traditional Unix-based file I/O interface. posix I/O follows a synchronous, blocking model by default, where each I/O operation (e.g., `read()`, `write()`) blocks the calling thread until the operation is completed. posix is widely used, as its synchronous nature makes it simple to use. However, the synchronous design also has downsides. In scenarios involving high concurrency, the performance and scalability of posix may suffer as multiple threads or processes are needed to parallelize I/O operations.

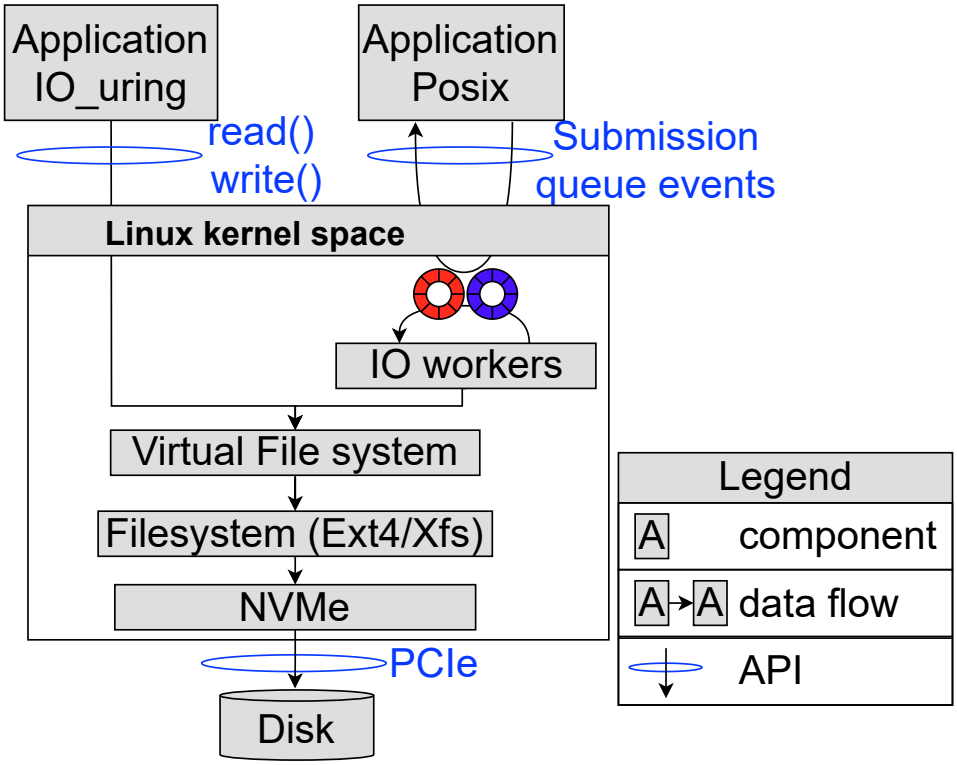


Figure 1: Control flow of posix and io_uring

2.2.2 libaio

libaio (Linux asynchronous I/O library) provides a user-space API for asynchronous I/O operations in Linux, not to be confused with posix aio. Unlike posix AIO, libaio offers an interface that enables applications to issue asynchronous read and write requests using syscalls. This is achieved through system calls such as `io_submit()` for submitting I/O requests and `io_getevents()` for retrieving the results of completed operations [10]. The events are queued up in kernel space, but the API was not always asynchronous as it promised. Additionally, does not support buffered IO.

2.2.3 io_uring

io_uring, introduced in the Linux kernel 5.1, provides a modern asynchronous I/O interface with reduced system call overhead. Two queues named the submission queue (SQ) and completion queue (CQ) are placed in the shared memory between the operating system (and thus the file system) and the application process. The application can then submit submission queue events (SQE), which the kernel will process on separate kernel worker threads. This allows I/O to be asynchronous, and to mitigate the context switch.

In this work we consider a configuration where Rocksdb is deployed on a Linux kernel using an ext4 or xfs file system. In figure 1 we show what the code paths look like from an architectural perspective, with the relavant details for performance.

In the figure, The left application uses posix to interface with the kernel, and the right

application uses io_uring to interface with the kernel. When the posix application does a read or write call, the posix code path immediately talks to the virtual file system, and its underlying stack whereas the io_uring application code path only submits its SQE to the queue and then returns. The kernel then has IO worker threads executing the request which then submit completion queue events (CQEs). The application checks the completion queue for completed events.

3 Design

To design our system, we use the Atlarge design process [11] and start by investigating their performance requirements. Our task is to assist in the development of faster key-value stores by making it easy to benchmark different storage APIs in RocksDB. We need to be able to easily switch between backends without recompilation as the large C++ project takes a long time to compile.

3.1 Functional Requirements

FR1 The system should support swapping storage engines within RocksDB, allowing users to select between at least POSIX and io_uring. A configuration option should be available to specify the desired storage engine (e.g., storage engine=POSIX).

FR2 Users should be allowed to perform benchmarks with different storage engines using RocksDB's db_bench tool with the new backends.

FR3 The backends should support all relavant storage APIs including io_uring posix and libaio.

FR4 The System benchmarks should be run on an emulated NVMe SSD for repeatable experiments.

FR5 The System should support instrumentation possibilities to allow users to dynamically instrument right before every write to disk and after every write to disk.

3.2 Design Overview

The design we implemented is shown in Figure 2. RocksDB's plugin system allows the registration of different file systems, enabling users to select custom file systems as RocksDB parameters. In our case, we register a file system called Engineswap (4), which acts as a pass-through layer. Engineswap forwards I/O calls to specific custom backends such as libaio (5), posix (7) and io_uring (6) (**FR3**) based on a parameter given to db_bench, the RocksDB internal benchmark. The backend used is determined based on a RocksDB configuration parameter or an environment variable.

3.3 Design Motivation

We chose this architecture because similar approaches, such as ZenFS [12] and DedupFS [13], have demonstrated success in using the plugin system for RocksDB, additionally, the plugin system provides the ability to select different storage backends **FR1**. It also supports db_bench out of the box (**FR2**).

Alternative designs, such as hijacking library calls, introduce potential instability and complexity, making debugging more challenging. Our design avoids these pitfalls and aligns directly with the requirements (**FR1**, **FR3**, and **FR4**). By building upon existing

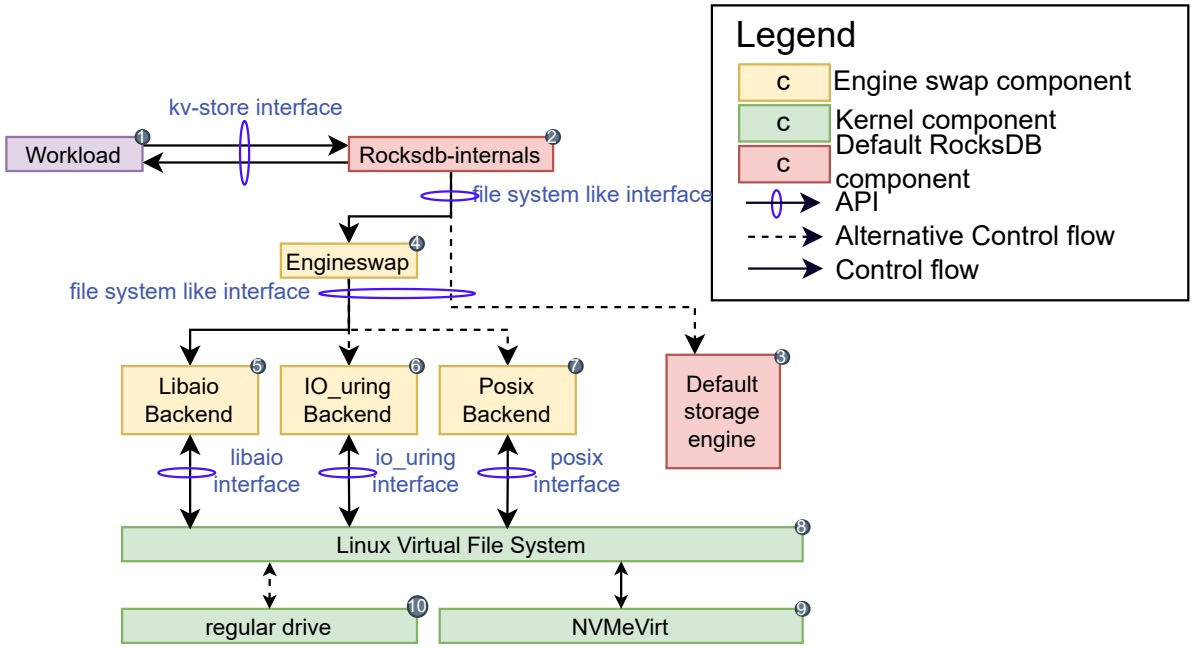


Figure 2: Design overview

plugin infrastructure, the architecture remains extensible for future experimentation with additional storage backends.

For user-level dynamic tracing on linux, USDT-probes are the only option supported by common tools like bpftrace that would satisfy **FR5**. The location of the probes was chosen to avoid mistakes in the experimentation phase of the project.

4 Implementation

In this section, we implement the engineswap plugin into a system. The implementation of the engineswap is explained in 4.1, and the implementation of the system around it is explained in 4.2.

4.1 The engineswap plugin

The engineswap plugin has taken great inspiration from zenfs [12] and dedupfs [13]. We built a pass through file system which can be enabled using the db_bench fs_uri parameter followed by :// and the desired backend. Because RocksDB is designed with synchronous IO, we used the implementation of the RocksDB posix file system as a base-class, and overrode some methods to replace the calls with io_uring. In addition, we used user-level probes (USDT probes) [14] per backend per read and write call, one before and one after calls.

For io_uring we used liburing [?], a library which provides a lot of helper functionality assisting with the development of io_uring applications. Additionally, we made rings

thread local, so the setup of the rings (and the creation of the kernel threads) would happen only a few times. We ensured the creation of the rings happens less than 20 times using print statements. For the submission queue polling, we configured the kernel thread to go to sleep after 5 seconds without incoming SQEs.

4.2 The System

As io_uring is only available on Linux, our back-ends will not work on any other operating system or storage stack unless they implement io_uring and libaio as well. We got an ubuntu 22.04 VM, running the 6.8.0 kernel provided by ubuntu.

The VM is running on 10 cores and 81 GB of memory of which we used 2 cores and 50 GB of memory for the virtual ssd. The host is running an Intel Xeon Silver 4416+ CPU.

We chose NVMeVirt for our virtual SSD, we use nvmevirt [15], it is more representative of real workloads over the alternative tmpfs as it has a real file system and configuration options to simulate other SSD speeds.

KF	Section	Experiment question	Workload
KF1	5.1	Is there a performance difference between engines?	real-world emulating
KF2	5.2	How does io_uring use CPU compared to posix?	Simple with WAL
KF3	5.3	Why is io_uring slower than posix?	simple without WAL

Table 1: Experiment overview

5 Experiments

In this chapter we find performance differences between io_uring and posix by experiments summarised in table 1. From these experiments, we conclude the following key findings:

KF1 Io_uring throughput and latency are worse than posix and libaio, and deteriorate even further when submission queue polling is enabled (Section 5.1). We ran a real world emulating workload and measured get() and put() latency. We compared storage engines between posix, io_uring, io_uring with submission queue polling and libaio. We found throughput and latency to be similar between libaio and posix, where throughput is increasingly worse for io_uring and io_uring with submission queue polling.

KF2 The time lost by Io_uring is mostly off-cpu time, and very little on cpu time. (Section 5.2). We run a much simpler workload consisting of SStable writes and WAL writes on posix and io_uring engines. We made a CPU profile and visualised these in flame graphs. We found very similar profiles between io_uring and posix, and concluded the latency and throughput difference not to be on the CPU.

KF3 When doing single-threaded I/O io_uring introduces some overhead with synchronisation, and a large increase in file system latency. (Section 5.3). We run a fully single-threaded workload without WAL writes and measure the latency of code path components for io_uring and posix. We plot these in a stack plot (5) which shows the io_uring worker application thread synchronisation overheads to be around 60 microseconds on average both ways. Additionally, we find an unexpected file system latency increase in io_uring compared to posix.

5.1 Experiment 1, A simple benchmark

In our first experiment, we wanted to answer *is there a difference in performance between our io_uring, libaio, and posix storage configurations?*.

5.1.1 Method

We ran a workload with similar characteristics provided by Cao et al. [16]. And measured (among other metrics) throughput and P99 latency of get() and put() calls. We chose

these metrics as Dean and Barroso noted in [17] that p99 outliers disproportionately affect overall system performance, often causing bottlenecks that degrade responsiveness for latency-sensitive applications.

5.1.2 Workload

[16] Found that key size in real-world workloads are typically small, with sizes ranging from 16 to 32 bytes, while the values vary from hundreds of bytes to a few kilobytes. Following these findings, the experiments in this project use key sizes of 16 bytes and 32 bytes, paired with values of 128 bytes, 512 bytes, and 1,024 bytes. These choices reflect a common distribution seen in many production systems, including metadata storage and log processing applications. By including these values, the benchmark tests both small and moderate value sizes, ensuring a comprehensive evaluation of storage engine performance across diverse workloads. Our parameters are defined in table 2.

Table 2: Workload 1: Customized MixGraph Workload

Parameter	**Value**	**Details**
Key Size	48 bytes	Fixed
Value Size Distribution	Log-normal	Mean: 268 bytes, StdDev: 25.45 KB
Key Distribution	Zipfian-like	Parameters: $a = 0.002312$, $b = 0.3467$
Key Range Distribution	Custom	Parameters: $a = 14.18$, $b = -2.917$, $c = 0.0164$, $d = -0.08082$
Number of Key Ranges	30	Fixed
Operation Ratios	83% Get, 14% Put, 3% Seek	Mixed operations
I/O Settings	Direct I/O	Reads, flush, compaction
Cache Size	256 MB	Fixed

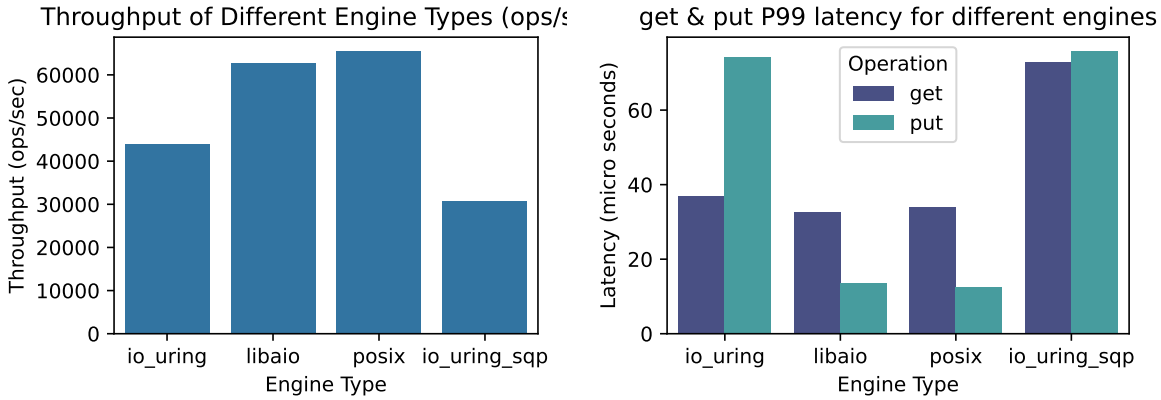


Figure 3: Throughput

5.1.3 Results of the simple benchmark

When examining the throughput performance of the four backends—POSIX, libaio, io_uring, and io_uring with polling—it is evident that the POSIX backend consistently outperformed the others, achieving the highest throughput of 65,467 and 118.6 MB/s operations per second (OPS). This performance was slightly better than libaio, which reached 62,590 OPS and 113.3 MB/s, showcasing its efficiency in handling synchronous I/O operations.

On the other hand, the io_uring backend, while leveraging modern asynchronous I/O mechanisms, achieved a significantly lower throughput of 44,013 OPS and 79.6 MB/s. Despite its design to reduce syscall overhead, io_uring lagged behind both POSIX and libaio, possibly due to being better optimized for single-threaded workloads.

The io_uring with polling backend demonstrated the lowest throughput among the four, achieving only 30,748 OPS and 55.7 MB/s. Although the addition of polling was intended to improve performance by minimizing latency, it seems to have introduced overheads that negatively impacted throughput in this single-threaded configuration.

To summarise, we used a workload emulating real world traces from [16] and measured get() and put() latencies between io_uring and posix. We conclude in **Key finding 1:** io_uring throughput and latency are worse than posix and libaio, and deteriorate even further when submission queue polling is enabled.

5.2 Experiment 2, CPU profile

In the benchmark experiment, we concluded that io_uring and libaio were both slower than posix. In this and the following experiments, we zoom in on the question: *How is io_uring CPU usage distributed over components compared to posix?* . To do this, we want a full picture of time spent during the database, so we chose a profiling technique, using perf.

5.2.1 Method

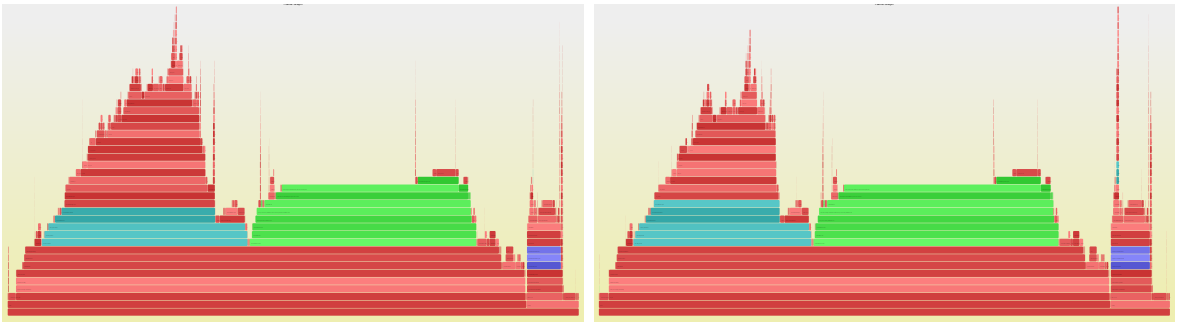
We configured perf to capture stack samples from our process' main thread at 1000hz while running our workload. If our main thread was not on cpu, perf discards the sample. We can then count per function the stacksamples associated with this function, and find which functions took a lot of CPU time.

We recompiled lib_uring from the source with the -fno-omit-framepointer flag to allow perf to unwind stacks properly. Additionally, we rebuild the file system on the virtual SSD between invocations of db_bench run to avoid fragmentation.

5.2.2 Workload

To speed up the experiment turnaround time, we made a much simpler workload which is easier to analyse. In the evaluation, we ensured that the same behaviour still exists in this workload. The workload itself consists of randomly putting 50.000.000 pairs into the database, without compactions, but we kept the write ahead log (WAL). This workload

is almost fully single-threaded which makes its performance much easier to analyse. It also sped up experiment runtime from $\tilde{8}$ hours to $\tilde{4}$ minutes.



(a) io_uring (b) posix

Figure 4: Comparison of flame graphs for io_uring and posix

5.2.3 Results

We plot our results in a flame graph in figures 4a for io_uring and 4b for posix. Functions are represented by horizontal bars, of which the width represents the size spent in this function. The order of different towers do not matter. The height on the stack represents function call graph depth.

The function names are omitted, but we annotate some sections of interest. The teal (left) tower are the WAL writes which is our area of interest as these happen with different storage engines. The green blob in the middle represent the memtable insertions, which do not use IO. On the right, the memtable flushes are marked in blue.

When analysing the different tower widths, we found a striking similarity between the two graphs, indicating the on-CPU time is similarly distributed between the engines. Moreover, from the stack samples captured for io_uring, 26.82% are appending the WAL file whereas for posix this is 22.32%, which is not as large a difference as we saw in the previous experiment as we would see the WAL io_uring tower to be close to 1.5 times wider than posix rather than a few per cent.

So because the time is distributed similarly between io_uring and posix we conclude the following in **Key finding 2:** *The time lost by io_uring is mostly off-CPU time, and very little on CPU time.*

5.3 Experiment 3, Latency analysis

In the previous experiment, we found that the extra logic employed by io_uring to offload to the worker takes a lot of CPU time, where we hypothesised that this could be the cause of the slowdown in io_uring compared to posix. However, the previous experiment only measured time running on the CPU, and not actual time spent between operations. Additionally, it did not measure time synchronising the worker and application thread and its effect on latency. In this experiment, we try to overcome these shortcomings by asking: *Where is the actual time spent in io_uring and Posix?*.

5.3.1 Method

To find the answer, we used a latency analysis technique leveraging bpftrace and our USDT probes from chapter 3. To Find the code path taken by the io_uring and posix syscalls, we used trace-cmd [18] and its function_graph tracer to list the functions called. Based on this graph, we created a bpftrace script which measured the time spent in a few sections by instrumenting different probes and calculating the time in nanoseconds between their hits. For posix, we identified the following sections:

1. **Time to go from user to kernel space**. This is measured as the difference in nanoseconds between our USDT probe before the syscall and a kprobe (instrumentation that triggers on kernel function enter) attached to vfs_write().

2. **Time for write preparation**, involves verifying the permissions of the files, resolving the file descriptor with the file structure, etc. This is measured with a kprobe attached to ext4_write_iter, and the previous event.

3. **File system latency**, which involves the time required for the file system to write to the file. This is measured with a kretprobe (instrumentation that triggers on kernel function exit), attached again to ext4_file_write_iter and xfs_file_write_iter and the previous event.

4. **Time to go from kernel space to user space**, we measure again with a USDT probe, this time after our syscall and the previous event.

For io_uring, we measured the extra io_uring logic as well as the old logic. We instrument io_uring as follows:

1. **Time to go from user to kernel space**. This is the same as in posix, but this time we instrument a kprobe attached to fdget() and the USDT probe from 3 before the preparation and submission of the SQE.

2. **SQE processing**, which involves verifying the permissions of the files, processing some flags, resolving file descriptors, etc. This is measured with a kretprobe attached to io_submit_sqes and the previous event.

3. **Io worker synchronisation**, Remember that io_uring uses worker threads to do the hard work (section 2.2.3) and there is the time required for the kernel worker to wake up or realise it has work to do. We measure this time with a krprobe attached to io_worker_handle_work and the previous event.

4. **Write preparation** This is similar to the posix preparation. This is again measured by a kprobe attached to ext4_write_iter and the previous event.

5. **File system latency**, this is the same as in posix. This is again measured with a kretprobe (instrumentation that triggers on kernel function exit), attached again to ext4_file_write_iter and xfs_file_write_iter and the previous event.

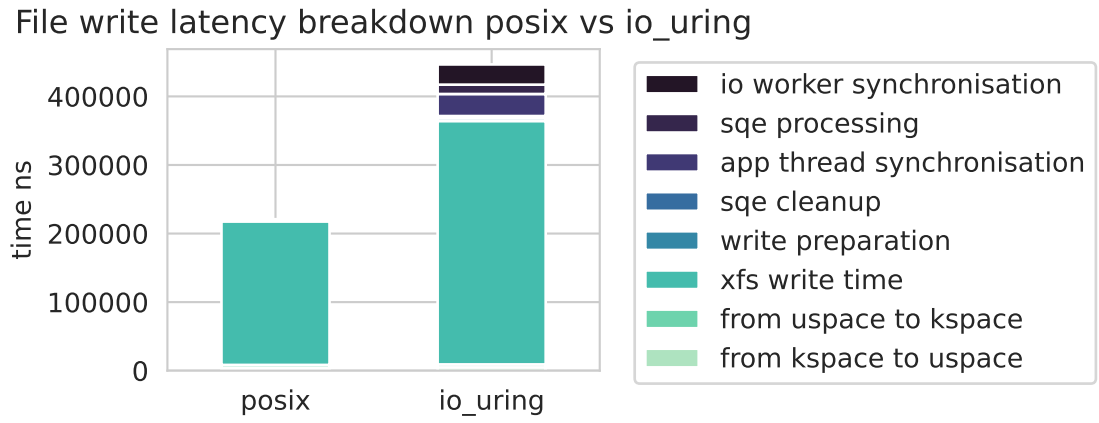


Figure 5: Summary of latency analysis. (Lower is better)

6. **SQE cleanup**. This is the time needed to create and submit the CQE. It is measured with a kretprobe attached to kiocb_done and the previous event.

7. **Application thread synchronisation**, is the time spent between submitting the CQE and the application thread realising the CQE has been submitted. We measure this kprobe attached to finish_wait and the previous event.

8. **Time to go from kernel space to user space**, we measure again with a USDT probe after our syscall and the previous event.

We verified that all of the instrumentation is always passed in order for both posix and io_uring. Additionally, we verified that the creation of the kernel worker thread occurs a very small amount of times (10 times for 50 million writes). We also rebuild the file system between every invocation of db_bench.

5.3.2 Workload

We use almost the same workload as in section 5.2 consisting of filling the database with 50.000.000 pairs without compactions. We did disable the WAL writes to make the workload fully singlethreaded, making our tracing instrumentation more reliable.

5.3.3 Results

We plot our results in figure 5, where the vertical axis represents the average time spent in within a write call in nanoseconds, where the differently colored areas represent time in our defined sections of the code path. We captured 6098 samples per engine. The entire call for posix takes on average 223278 nanoseconds where io_uring takes 450546 nanoseconds. The totals are measured with a separate calculation to cross-reference the stack plot. We also looked at [19] where the auhors found file system latencies on hundreds of microseconds.

When analysing the plot, we notice the time spent synchronising between the io_uring worker and the application thread is significant. The time between the application submitting the request and the worker realising it has work to do, is on average 29 microseconds, where the reverse transaction is on average 32 microseconds.

Addressing the elephant in the plot is the large difference in file system latency, where io_uring spent on average 355 microseconds, and posix 209 microseconds. This is quite odd as the file system is a shared component between io_uring and posix. We tested both ext4 and xfs, obtaining similar results. We still suspect a performance or configuration bug to be the cause. To properly assess the cause behind our result, future work is needed.

To summarise, we used latency analysis with bpftrace to find latencies associated with different components. We conclude in **Key finding 3:** *When doing single-threaded I/O io_uring introduces some overhead with synchronisation, and a large increase in file system latency.*

6 Conclusion

RocksDB is an important piece of computing infrastructure deployed in many other projects where performance is important. We investigate if RocksDB can benefit when changing its engine to io_uring or other asynchronous APIs, even though it is designed for synchronous IO. To do this, we designed, implemented, and evaluated engineswap, a RocksDB plugin able to pass calls through the file system to different APIs including posix, io_uring and libaio. We find posix and libaio to have better throughput and latency compared to io_uring and in both real-world emulating, and simple synthetic workloads. However, we have reasons to believe performance bugs exist in our system and more work is needed to find why io_uring performs worse.

6.1 Limitations

Some limitations are enumerated below. At the time of writing, we really wants to continue with other projects and courses and avoid feature creep, causing us to cut quite importand features from the project.

No off-CPU flame graph

To accompany our CPU profile in section 5.2, an Off cpu flame graph would show much more insight into the difference between io_uring and posix performance. This would crosreference our latency analysis in 5.3 as well. Unfortunately off-CPU flame graphs are usually constructed using offcputime [20] which uses a kprobe attached to a function which got inlined in all the kernels we had access to. Where a recompilation of our own kernel did not fit on the VM provided and increasing the size of the virtual disk was deemed very risky and a little rude to our supervisors.

No crossreference for the file system latency increase in section 5.3

In **KF3** we found a large increase in file system latency between io_uring and posix. The increase in file system latency is rather alarming and signalling a performance bug that could jeopardise all our results so far. A crossreference to this result would be quite useful.

No error bars in the experiment from section 5.1

The experiment from **KF1** uses a real-world emulating workload to find the difference between io_uring and posix. Even though the experiment takes 8 hours to complete and gave a very large number of samples, we received advice that db_bench performance can vary between runs. Running the same experiment some more times should assist in solving this issue.

6.2 Suggestions for Future Research

Future research could figure out why file system latency is so much worse in io_uring compared to posix. To do this, it could attempt latency analysis on the file system to find where the latency exactly is coming from. It also could attempt to use bpftrace and trace-cmd to compare file system code path or use bpftrace to compare arguments fed to ext4_file_write_iter between the two engines. This research could lead to finding a performance bug in liburing or the Linux kernel.

7 References

[1] M. Platforms, "Rocksdb," accessed: 2024-11-28. [Online]. Available: https://github.com/facebook/rocksdb

[2] Y. Matsunobu, S. Dong, and H. Lee, "Myrocks: Lsm-tree database storage engine serving facebook's social graph," *Proc. VLDB Endow.*, vol. 13, no. 12, p. 3217–3230, Aug. 2020. [Online]. Available: https://doi.org/10.14778/3415478.3415546

[3] P. LLC, "Percona server for mongodb 6.0." [Online]. Available: https://docs.percona.com/percona-distribution-for-mongodb/

[4] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss, P. Bardea, A. Ranade, B. Darnell, B. Gruneir, J. Jaffray, L. Zhang, and P. Mattis, "Cockroachdb: The resilient geo-distributed sql database," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1493–1509. [Online]. Available: https://doi.org/10.1145/3318464.3386134

[5] H. van der Breggen, "Mysql customer: Booking.com." [Online]. Available: https://www.mysql.com/customers/view/?id=901

[6] D. Pacheco, "110 - cockroachdb for the control plane database / rfd / oxide," Aug. 2024. [Online]. Available: https://rfd.shared.oxide.computer/rfd/0110

[7] Z. Ren and A. Trivedi, "Performance characterization of modern storage stacks: Posix i/o, libaio, spdk, and io_uring," in *Proceedings of the 3rd Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems*, ser. CHEOPS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 35–45. [Online]. Available: https://doi.org/10.1145/3578353.3589545

[8] F. Engineering, "Rocksdb: A persistent key-value store for flash and ram storage," *Facebook Open Source Blog*, 2013, available at https://github.com/facebook/rocksdb.

[9] H. Kim and S. Lee, "Comparative analysis of modern storage engines: io_uring vs libaio," *ACM Transactions on Storage*, vol. 19, no. 2, pp. 1–18, 2023.

[10] H. Tao and Z. Yu, "Performance evaluation of linux asynchronous i/o," in *2008 International Conference on Computer Science and Software Engineering*. IEEE, 2008, pp. 100–105.

[11] A. Iosup, L. Versluis, A. Trivedi, E. van Eyk, L. Toader, V. van Beek, G. Frascaria, A. Musaafir, and S. Talluri, "The atlarge vision on the design of distributed systems and ecosystems," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, 2019, pp. 1765–1776.

[12] "Zenfs: A file system for rocksdb on zoned block devices," accessed: 2024-11-28. [Online]. Available: https://github.com/westerndigitalcorporation/zenfs

[13] "Dedupfilesystem," accessed: 2024-12-18. [Online]. Available: https://github.com/ajkr/dedupfs

[14] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic instrumentation of production systems," *Usenix Annual Tech Conference*, vol. 4. [Online]. Available: https://www.usenix.org/conference/2004-usenix-annual-technical-conference/dynamic-instrumentation-production-systems

[15] S.-H. Kim, J. Shim, E. Lee, S. Jeong, I. Kang, and J.-S. Kim, "NVMeVirt: A versatile software-defined virtual NVMe device," in *21st USENIX Conference on File and Storage Technologies (FAST 23)*. Santa Clara, CA: USENIX Association, Feb. 2023, pp. 379–394. [Online]. Available: https://www.usenix.org/conference/fast23/presentation/kim-sang-hoon

[16] Z. Cao, Y. Yu, Y. Qiu, Y. Hu, P. Lian, and G. Hu, "Tbe: A tensor-based key-value cache design on persistent memory," in *18th USENIX Conference on File and Storage Technologies (FAST)*, 2020, pp. 133–147. [Online]. Available: https://www.usenix.org/system/files/fast20-cao_zhichao.pdf

[17] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[18] S. Rostedt, "rostedt/trace-cmd," Dec. 2024. [Online]. Available: https://github.com/rostedt/trace-cmd

[19] H. Duan, L. Shi, Q. Zhuge, E. H.-M. Sha, C. Li, and Y. Liang, "An empirical study of nvm-based file system," in *2021 IEEE 10th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, 2021, pp. 1–6.

[20] B. Gregg, "offcputime-bpfcc(8) — bpfcc-tools — debian unstable — debian manpages." [Online]. Available: https://manpages.debian.org/unstable/bpfcc-tools/offcputime-bpfcc.8.en.html

A db_bench parameters

Keyfinding 1, the benchmark

```
./db_bench --benchmarks=mixgraph,stats,levelstats
--fs_uri=engineswap://$ENGINE --compression_type=none -histogram
--db=/local/nvmevirt
--use_existing_db=1
--cache_size=268435456
--key_dist_a=0.002312 -key_dist_b=0.3467
--keyrange_dist_a=14.18
--keyrange_dist_b=-2.917
--keyrange_dist_c=0.0164
--keyrange_dist_d=-0.08082
--keyrange_num=30
--value_k=0.2615
--value_sigma=25.45
--iter_k=2.517
--iter_sigma=14.236
--mix_get_ratio=0.83
--mix_put_ratio=0.14
--mix_seek_ratio=0.03
--sine_mix_rate_interval_milliseconds=5000
--sine_a=1000
--sine_b=0.000073
--sine_d=4500
--perf_level=2
--reads=420000000
--num=50000000
--key_size=48
--statistics=1
--stats_interval_seconds=1
--report_interval_seconds=60
--report_file=/home/user/simple-bench-out-$ENGINE.csv
```

A.0.1 Keyfinding 2, flamegraphs

```
$ROCKSDB_PATH/db_bench --benchmarks=fillrandom
--fs_uri=engineswap://$ENGINE
--db=/local/nvmevirt/
--compression_type=none -histogram
--disable_auto_compactions
--max_background_compactions=0
-cache_size=0
--num=50000000
```

A.0.2 Keyfinding 3, latency analysis

```
$ROCKSDB_PATH/db_bench --benchmarks=fillrandom
--fs_uri=engineswap://$ENGINE
    --db=/local/nvmevirt/
    --compression_type=none -histogram
    --disable_auto_compactions
    --max_background_compactions=0
    -cache_size=0
    --num=50000000
    --disable_wal=1
```

B Full flamegraphs

B.1 io_uring

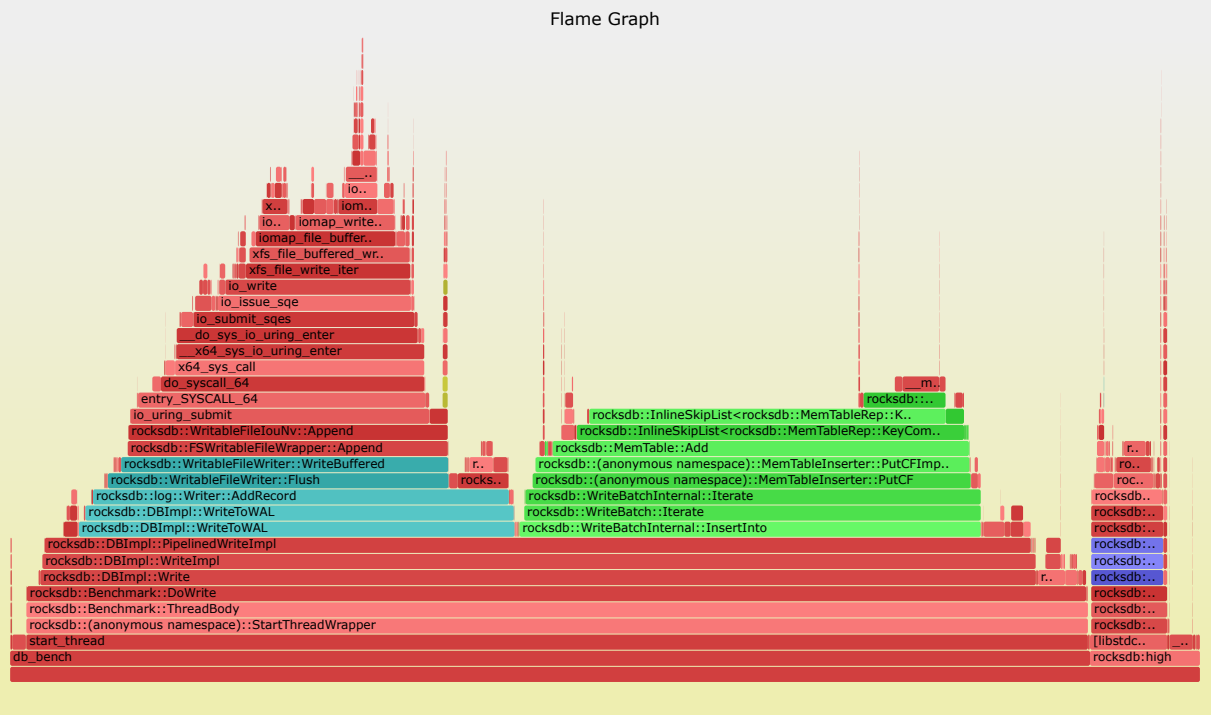


Figure 6: Full flamegraph for io_uring

B.2 posix

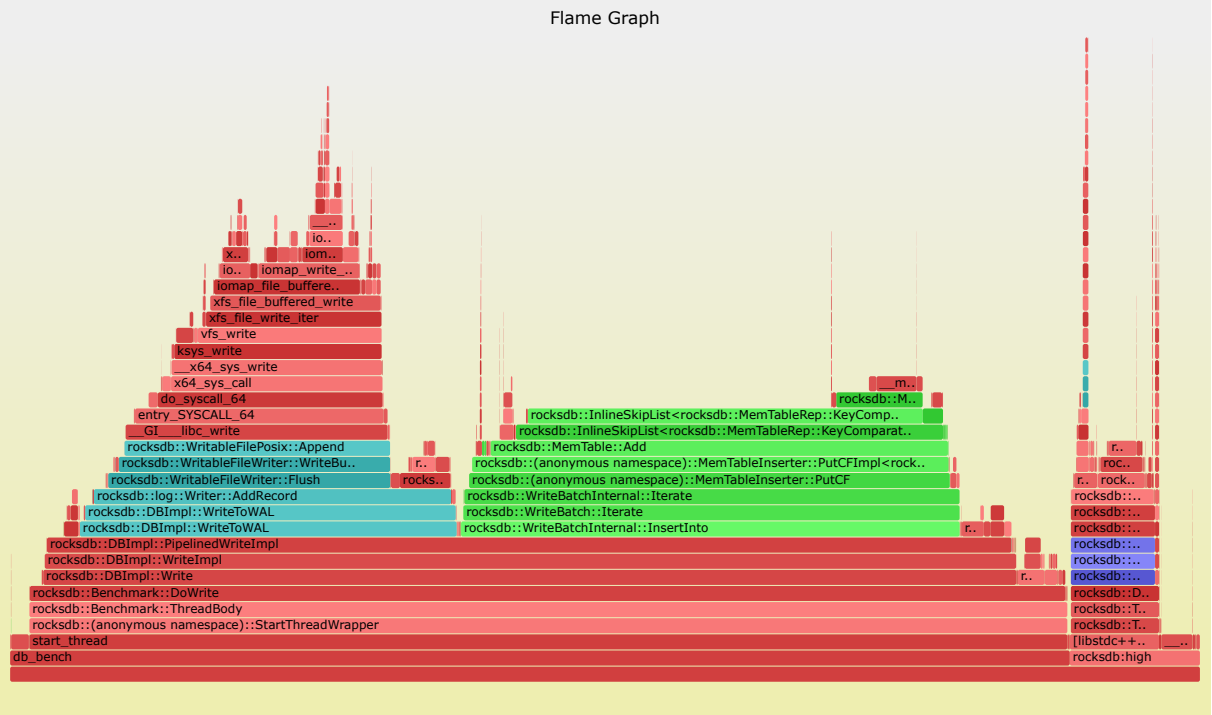


Figure 7: Full flamegraph for posix